Annales UMCS Informatica AI 4 (2006) 291-303

Annales UMCS
Informatica
Lublin-Polonia
Sectio AI

http://www.annales.umcs.lublin.pl/

Polymorphism – prose of Java programmers

Zdzisław Spławski*

Institute of Computer Science, Wrocław University of Technology,
Wybrzeże Wyspiańskiego 27, 50-370 Wrocław, Poland

Abstract

In Java programming language as implemented in JDK 5.0 there appear rather advanced kinds of polymorphism, even if they are hidden under different names. The notion of polymorphism unifies many concepts present in typed programming languages, not necessary object-oriented. We briefly define some varieties of polymorphism and trace them in Java. Java shows that "industrial" programming languages are able to express more abstract patterns using rather involved theoretical means, hence the "working programmer" has to be better educated in order to understand them, recognize them in different programming languages under different names and superficial syntax, and make good use of them.

Monsieur Jourdain. Par ma foi! il y a plus de quarante ans que je dis de la prose sans que j'en susse rien, et je vous suis le plus obligé du monde de m'avoir appris cela.
Molière

1. Introduction

Polymorphism $\langle \text{gr.} \pi o \lambda \upsilon \varsigma = \text{many} + \mu o \rho \varphi \eta = \text{form} \rangle$ in general means "multiform" and allows the same code to be assigned multiple types. This may be achieved in many ways, hence there are varieties of polymorphism in computer science and still its new kinds are being proposed, but most of them are known only to theoreticians. The unqualified term "polymorphism" may cause some confusion, since among programmers it is often used to mean concrete kinds of polymorphism. For object-oriented programmers it almost always means inclusion polymorphism, for functional programmers – shallow parametric polymorphism, for theoreticians it usually means impredicative parametric polymorphism, as used in System F (called also polymorphic or second-order lambda-calculus). The "working programmer" usually identifies this term with inclusion polymorphism, but it does not mean that he did not

* *E-mail address*: zdzislaw.splawski@pwr.wroc.pl

unconsciously use other kinds of polymorphism (*vide* monsieur Jourdain, who all of his life has spoken prose not being aware of this fact). Our goal is to disclose other kinds of polymorphism which may be found in Java in order to make these mechanisms more accessible to the working programmer.

For a long time people have discussed how to include in Java programming language parametric polymorphism which allows abstracting over types. Some proposals were considered, among them GJ [1]. However, it turned out that importing this mechanism from the functional programming languages, e.g. Standard ML, raises some problems in connection with inclusion polymorphism. Thorup and Torgersen [2] proposed a variant of bounded polymorphism which integrates the merits of virtual types with F-bounded polymorphism. Their type system has been further developed, formalized, and proven sound by Igarashi and Viroli [3] within the Featherweight GJ calculus [4]. This mechanism in now available in JDK 5.0. Its short description can be found e.g. in [5,6].

2. Varieties of polymorphism

Many kinds of polymorphism can be found in modern programming languages. Below we provide necessary definitions (somewhat simplified, but suitable for this paper) and briefly characterized polymorphisms discussed in this paper.

Objects are programming units that associate data (called instance variables) with the operations (called methods) that can use or affect these data.

Classes are extensible templates for creating objects, providing initial values for instance variables and the bodies for methods. New objects can be created from a class with the `new` operator.

In terms of implementation we can recognize *universal polymorphism* when the same code is executed for any admissible type, whereas in the case of *ad-hoc polymorphism* a different code is executed for each type. There are two major kinds of universal polymorphism: *parametric* and *inclusion polymorphism*, and two major kinds of ad-hoc polymorphism: *coercion* and *overloading*.

Parametric polymorphism allows a simple piece of code to be typed "generically", using variables instead of actual types. These type variables are instantiated with concrete types. Parametric polymorphism guarantees uniform behavior in the range of types.

In *inclusion polymorphism* an object can be viewed as belonging to many different classes that need not to be disjoint; that is, there may be inclusion of classes. Inclusion polymorphism models *subtyping* and *subclassing* (*inheritance*).

Subtyping. The type of an object is just the set of names and types of its methods. We say type `S` is a *subtype* of `T` (written `S <: T`), if a value of type `S` can be used in any context in which a value of type `T` is expected. We say `T` is a *supertype* of `S` if `S` is a subtype of `T`. Subtyping relation should be reflexive and

transitive. Because values may have multiple types in languages supporting subtyping, we say these languages support *subtype polymorphism*. Object types fit naturally into the subtype relation.

Subclassing (*inheritance*). Classes are used not only to create objects, but also to create richer *subclasses* by inheritance that allows new classes to be derived from old ones by adding implementations of new methods or *overriding* (i.e. replacing) implementations of old methods. Subclass relation is defined analogically to the subtype relation. We will write `S <: T` to mean also that class `S` is a *subclass* of `T`, which is a *superclass* of `S`. The meaning of "`<:`" should be clear from the context. The reference to an object of a subclass can be used anywhere that a reference to an object of its superclass is expected.

In general we have two hierarchies, one induced by inheritance, the other one corresponding to the subtyping relation. These two hierarchies are, in principle, completely distinct [7], e.g. in Objective Caml [8], but in many typed object-oriented languages the two hierarchies coincide. In old Java (without generic types) the terms *subtype* and *subclass* were basically interchangeable, but in Java 5.0 this relationship is more complicated, as we shall see later.

Bounded polymorphism integrates parametric and subtype polymorphism, and allows restriction on type variables by specifying upper and/or lower bounds.

Ad-hoc polymorphism allows to use the same name for different piece of code that may behave in unrelated ways for each type.

Coercion is a semantic operation that converts an argument to the expected type in a situation that would otherwise result in a type error. The special well-known case of coercion is *promotion*.

In *overloading* the same name is used to denote different functions or methods and the context is used to decide which function or method is denoted by a particular instance of the name.

In languages with subtyping we differentiate at least two disciplines of method (or function) selection:

– *early binding* is based on static (compile-time) type information;
– *late binding* is based on dynamic (run-time) type information.

Classification of type systems can be found in [9], see also [10]. Theoretical foundations of type systems in programming languages are contained in monographs [11-14].

3. Polymorphism "ad hoc"

There are two major kinds of ad-hoc polymorphism: *overloading* and *coercion*, but the boundary between them in many cases is not sharp and depends on the implementation.

3.1. Operator overloading

Operators for arithmetical operations in Java are overloaded for all numeric types and strings, but both arguments must be of the same type. In the expression '5+3.4' operator '+' denotes addition of two numbers of type double and left argument is coerced to the type double. In the expression '5+3.4F' two numbers of type float are added with appropriate coercion of the left argument. Of course different object code is generated by the compiler in both cases. However, one may imagine that operator '+' is overloaded for four combinations of argument types (int and double). The same situation may be observed for other arithmetic operators.

In this example we may consider the same expression to exhibit overloading or coercion (or both), depending on implementation decision.

3.2. Method overloading

Most object-oriented programming languages, including Java, allow programmers to overload method names on classes. A method name is overloaded in a context if it is used to represent two or more distinct methods, and the method represented by the overloaded name is determined by its signature, like the method name m from the class C defined below. As long as their signatures are different, Java treats methods with overloaded names as though they had completely different names. The compiler statically (using early binding) determines what method code is to be executed.

```
public class C {
   void m( ) { System.out.print("C.m() "); }
   void m( C other ) { System.out.print("C.m(C) "); }
}
```

In Java, the overloading can happen when a method in a superclass is inherited in a subclass that has a method with the same name, but different signature.

```
public class SubCOverload extends C {
   void m( SubCOverload other ) // overloading m
   { System.out.print("SubCOverload.m(SubCOverload) "); }
}
```

Class SubCOverload has three overloaded methods with name m.

```
public class TestOverload {
   public static void main(String[] args) {
      C c = new C();
      C c1 = new SubCOverload();
      SubCOverload sc = new SubCOverload();
```

```
                                        System.out.print(" c: ");
    c.m(); c.m(c); c.m(c1); c.m(sc);    System.out.print("\nc1: ");
    c1.m(); c1.m(c); c1.m(c1); c1.m(sc); System.out.print("\nsc: ");
    sc.m(); sc.m(c); sc.m(c1); sc.m(sc); System.out.println();
```

Executing code of the class TestOverload we obtain the following result.

```
 c: C.m() C.m(C) C.m(C) C.m(C)
c1: C.m() C.m(C) C.m(C) C.m(C)
sc: C.m() C.m(C) C.m(C) SubCOverload.m(SubCOverload)
```

It illustrates the fact, that the class SubCOverload has three overloaded methods m, and that Java uses early binding for overloaded methods.

3.3. Coercion between primitive types and wrapper classes

The type of an object is just the set of names and of its methods. In Java these types are called *reference types* (since objects in Java are accessed exclusively by references). But Java has also *primitive types* like int, double, or boolean. Values of these types are not objects, but for each primitive type there exists wrapper class, e.g. Integer, Double, Boolean, which can be used in contexts where primitive types are not allowed.

Converting between primitive types, like int or boolean, and their wrapper classes like Integer and Boolean was very annoying in old Java. Unfortunately, these back and forth conversions could not be avoided since only objects can be stored in collections. Below the essential part of the wrapper class Integer is shown.

```
public final class Integer {
   private int i;
   public Integer(int i) { this.i = i; }
   public int intValue() { return i; }
```

The following code illustrates the operations of "wrapping" and "unwrapping" integer value.

```
   Integer wi1 = new Integer(5); // wrapping
   int i1 = wi1.intValue(); // unwrapping
```

Using autoboxing/unboxing mechanism the code is much more concise and easier to follow. This is an example of coercion, hence ad-hoc polymorphism.

```
   Integer wi2 = 5; // autoboxing
   int i2 = wi2; // unboxing
```

4. Polymorphic classes

4.1. Inclusion polymorphism

Suppose we want to define a class which provides the basic functionality of a pair without regard for specific types. In old Java this could have been done using inclusion polymorphism. We had to define a pair whose both elements are instances of the Object class, since every class in Java inherits from it, hence every object can be stored in such a pair.

```
public class ObjectPair {
   private Object e1;
   private Object e2;

   public ObjectPair(Object e1, Object e2)
   { this.e1 = e1; this.e2 = e2; }
   public Object getFst()  { return e1; }
   public Object getSnd() { return e2; }
   public String toString() { return "(" + e1 + ", " + e2 + ")"; }
}
```

Unfortunately, when we use this class, downcasts (with time and memory overhead) are required.

```
ObjectPair p;
p = new ObjectPair("Five", new Integer(5));

String numeral = (String) p.getFst();
Integer number = (Integer) p.getSnd();

p = new ObjectPair(new Integer(5), "Five");
numeral = (String) p.getFst(); // throws ClassCastException
```

Java compiler generates code, which checks dynamically correctness of downcasting operation and possibly throws ClassCastException. This affects program efficiency.

Method overriding with late binding exhibits quite different behavior from that of the method overloading with early binding. When "object-oriented working programmer" speaks about polymorphism, he usually refers to this mechanism. In Java, when a method is redefined in a subclass with exactly the same signature as the original method in the superclass then we have overriding and the binding of method calls occurs at run time (late binding). If the new method has the same name, but different signature, then we have overloading as

described in Subsection 3.2. and the binding occurs at compile time (early binding).

In the example below both classes SubC1Override and SubC2Override inherit from the class C, defined in Subsection 3.2, and they override (redefine) inherited method m.

```
public class SubC1Override extends C {
   void m( ) { System.out.print("SubC1Override.m() "); }
}
public class SubC2Override extends C {
   void m( ) { System.out.print("SubC2Override.m() "); }
}
```

In the code below the list of instances of these classes is traversed, and for each object its method m is invoked. This method prints the name of a class in which its body has been defined. LinkedList is a standard collection, which contains instances of class Object. We need downcasting (C)i.next() to make use of method m. This is a typical program with collections and inclusion polymorphism.

```
import java.util.*;
public class TestOverrideIncl {
   public static void main(String[] args) {
      LinkedList l = new LinkedList();
      l.add(new C());
      l.add(new SubC1Override());
      l.add(new SubC2Override());
      for (Iterator i = l.iterator(); i.hasNext(); ) {
         ( (C)i.next()).m();
      }
      System.out.println();
   }
}
```

Program output is given below. Note, that the code of the method m comes from classes objects are instantiated, and not necessary from class C, to which they were downcast. This proves that late binding was used (in C++ terminology these methods are virtual).

```
C.m() SubC1Override.m() SubC2Override.m()
```

Unfortunately, downcasting is always dangerous, since there is no guarantee that all objects in the collection are instances of class C (or its subclasses).

4.2. Parametric polymorphism

A class for a pair can be defined using parametric polymorphism (or generic class in Java terminology).

```
public class Pair<A, B> {
   private A e1;
   private B e2;

   public Pair(A e1, B e2)
   { this.e1 = e1; this.e2 = e2; }

   public A getFst()   { return e1; }
   public B getSnd() { return e2; }
   public String toString() { return "(" + e1 + ", " + e2 + ")"; }
```

Below this class is used to create a pair for strings and integers. In Java generic class can be instantiated with reference types only. We do not need downcasting. All type checking is done statically during compilation and ClassCastException will never be thrown.

```
Pair<String, Integer> p;
p = new Pair<String, Integer> ("Five", 5);

String numeral = p.getFst();
Integer number = p.getSnd();
```

The code below illustrates the advantages of Java generics over the code which uses inclusion polymorphism (class TestOverrideIncl from the previous section). Again, there are no casts. Notice also shorter form of the for loop.

```
import java.util.*;
public class TestOverrideParam {
   public static void main(String[] args) {
   LinkedList<C> l = new LinkedList<C>();
   l.add(new C());
   l.add(new SubC1Override());
   l.add(new SubC2Override());
      for (C e:l) { // read: for each e of type C in l
      e.m();
   }
   System.out.println();
   }
}
```

This is a typical program with collections and parametric polymorphism.

4.3. Integrating parametric and inclusion polymorphism

Java 5.0 integrates parametric and inclusion polymorphism as illustrates class PairMut. It extends functionality of class Pair with two new methods, which allow to change objects stored in a pair.

```
public class PairMut<A, B> extends Pair<A,B>{
   public PairMut(A e1, B e2) { super(e1, e2); }
   public void setFst(A e) { e1 = e; }
   public void setSnd(B e) { e2 = e; }
}
```

5. Implementation of generics

When generating bytecode for a generic class, Java compiler replaces type parameters by their *erasure*. Basically, erasure gets rid of (or *erasures*) all generic type information, and generates *one* bytecode for a generic class, which contains nothing but ordinary types, and which is executed for each instantiation of this class. This guarantees backward compatibility with legacy code. For an unbounded type parameter its erasure is Object. For an upper-bounded type parameter its erasure is the erasure of its upper bound.

With the addition of generics, the relationship between subtyping and subclassing has become more complex. In the code below (references to) objects pS and pI have distinct types (Pair<String, String> and Pair<Integer, Integer>, respectively), but are instances of the same class Pair, which is called *raw type*. All reference types in old Java were raw types in this terminology and they are legitimate in Java 5.0.

```
Pair<String, String> pS = new Pair<String, String>("fst","snd");
Pair<Integer, Integer> pI = new Pair<Integer, Integer>(1,2);
System.out.println(pS.getClass());       // prints: class Pair
System.out.println(pI.getClass());       // prints: class Pair
System.out.println(pS instanceof Pair); // prints: true
System.out.println(pI instanceof Pair);  // prints: true
```

Consequently, the expression pS instanceof Pair<String, String> is illegal and gives rise to the compilation error *"illegal generic type for instanceof"*.

This erasure implementation enforces some limitations on generics in Java, e.g. type variables in parametric polymorphism cannot be instantiated with primitive types. This limitation is not serious in the presence of autoboxing/ unboxing mechanism.

For comparison, templates in C++ are generally not type checked until they are instantiated. They are typically compiled into disjoint code for each

instantiation rather then a single code, hence problems arise with code bloat, but type variables *can* be instantiated with primitive types.

6. Generic types are not covariant

We say that type operator G is *covariant* (or that subtyping is covariant for G) if S <: T implies G<S> <: G<T>. We call it is *contravariant* if S <: T implies G<T> <: G<S>. We say that G is *invariant* if the conjunction of S <: T and T <: S implies G<S> <: G<T>.

Subtyping is invariant for generic types. We show only why it cannot be covariant. Suppose we want to have a method that prints elements of any pair. Here is a naive attempt using generics:

```
public static void printPairOfObjects(Pair<Object,Object> p) {
   System.out.println( "(" + p.getFst() + ", " + p.getSnd() + ")");
}
```

The problem is that it only works for Pair<Object,Object> which is *not* a supertype of all pairs! Compiling the following code we obtain compilation errors in lines 2 and 3.

```
PairMut <String, Integer> pSI;
pSI = new PairMut <String, Integer> ("Five", 5); // 1
printPairOfObjects(pSI);                         // 2
PairMut<Object, Object> pOO = pSI;               // 3: incompatible types
```

Suppose that assignment in line 3 was accepted. Then the following instructions must also be accepted.

```
pOO.setFst(new Object());  // 4
String s = pSI.getFst();   // 5
```

Accessing pair pSI through the alias pOO arbitrary object can be inserted to the pair, as was done in line 4. Now in line 5 we attempt to assign an Object to a String! For the same reason line 2 is illegal.

The argument against covariant subtyping for generic classes also applies to arrays, but Java actually permits covariant subtyping of arrays. This feature is now generally considered a flaw in the language design, since it seriously affects the performance of programs involving arrays. The reason is that the unsound subtyping rule must be compensated with a run-time check on *every* assignment to *any* array, to make sure the value being written belongs to the actual type of the elements of the array. If this type checking fails, the ArrayStoreException is thrown. In the example below we refer to classes C and SubCOverload defined in Subsection 3.2. Assignment arrC = arrSC is legal, because of covariant subtyping: SubCOverload <: C hence SubCOverload[] <: C[].

```
SubCOverload[] arrSC = new SubCOverload(), new SubCOverload();
C[] arrC = arrSC;
arrC[0] = new C();
arrSC[0].m(new SubCOverload()); // throws ArrayStoreException
```

Since arrays in Java are covariant, but generic types are invariant, one cannot create an array of a generic type using named variables (but unbounded wildcards are allowed). The declaration:

```
Pair <Integer, Integer>[] iPArr = new Pair <Integer, Integer>[5];
```

is illegal and causes compilation error *"generic array creation"*.

7. Wildcards and bounded polymorphism

The supertype of all kinds of pairs is `Pair<?,?>`, where wildcard '`?`' stands for some unknown type. Here is the code for our printing method.

```
public static void printPair(Pair<?,?> p) {
   System.out.println("(" + p.getFst() + ", " + p.getSnd() + ")");
}
```

Syntactically, a wildcard is an expression of the form '?', possibly annotated with an upper bound, as in '`? extends T`', or with a lower bound, as in '`? super T`'. As demonstrated in Section 6 subtyping is *invariant* for parameterized types. Using wildcards one may also express *covariant* and *contravariant* subtyping. Parameterized types with `extends`-bounded wildcards give rise to covariant subtyping: if `A<:B` then `G<? extends A> <: G<? extends B>`. Dually, `super`-bounds give rise to contravariant subtyping: if `A<:B` then `G<? super B> <: G<? super A>`.

Similarly one can specify upper bounds for named type parameters '`S extends Foo`'. In the absence of a type bound, a type parameter is assumed to be bounded by `Object`. Only wildcards can have lower bounds. A named type parameter can have more than one upper bound, but a wildcard can have only a single (upper or lower) bound. A formalization of wildcards is described in [15].

8. Polymorphic methods

Methods can also be made generic, independently of the class in which they are defined by adding a list of formal type arguments to its definition. Suppose we want to write static utility method `max`, which returns greater of its two arguments. Static methods are outside the scope of class-level type parameters (this is another limitation caused by erasure semantics), so we have to use generic method, parameterized by a type of its argument. But arguments must be comparable, hence the type must specify appropriate method for comparison, which can be assured by a bound.

Java standard library contains the generic interface Comparable<T>.

```
public interface Comparable<T> {
   public int compareTo(T o);
}
```

It specifies a method that compares 'this' object with the specified object for order.

Here is our polymorphic method we were looking for:

```
public class Test{
   public static <T extends Comparable<T>> T max(T e1, T e2) {
      return e1.compareTo(e2) > 0 ? e1 : e2;
   }
   public static void main(String[] args) {
      System.out.println(max("Adam", "Dick"));
      System.out.println(max(6, 2));    // autoboxing to Integer
   }
}
```

Notice, that when extends is used to denote a type parameter bound, it does not denote a subclass-superclass relationship, but rather a subtype-supertype relationship.

9. Conclusions

The notion of polymorphism unifies many concepts present in typed programming languages. We briefly defined some varieties of polymorphism and traced them in Java 5.0. Java shows that "industrial" programming languages are able to express more abstract patterns using rather involved theoretical means, hence the "working programmer" has to be better educated in order to understand them, recognize them in different programming languages under different names and make good use of them. Programmers are often unpleasantly surprised when a mechanism exhibits different behavior in another language, or if different names turn out to denote the same mechanism.

It would be also interesting to compare polymorphic features of some other "industrial" programming languages (e.g. C++, C#) from more abstract perspective than that used in manuals or tutorials. "Working programmer" should have a command of many programming languages which quickly evolve, and deeper understanding of underlying concepts would greatly alleviate his task.

References

[1] Bracha G., Odersky M., Stoutamire D., Wadler P., *Making the future safe for the past: Adding genericity to the Java programming language*. In: Proceedings of ACM Symposium on

Object-Oriented Programming: Systems, Languages and Applications (OOPSLA), ACM Press, (1998) 183.
[2] Thorup K.K., Torgersen M., *Unifying genericity. Combining the benefits of virtual types and parameterized classes*. In: Proceedings of European Symposium on Object-Oriented Programming (ECOOP). LNCS **,** Springer-Verlag, (1999) 186.
[3] Igarashi A., Viroli M., *On variance-based subtyping for parametric types*. In: Proceedings of European Symposium on Object-Oriented Programming (ECOOP). LNCS **,** Springer-Verlag, (2002) 441.
[4] Igarashi A., Pierce B., Wadler P., *Featherweight Java: A minimal core calculus for Java and GJ*. In: Proceedings of ACM Symposium on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA), ACM Press, (1999) 132.
[5] Bracha G., *Generics in the Java programming language*. (2004) http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf
[6] Torgersen M., Ernst E., Plesner Hansen C., Ahé P.v.d., Bracha G., Gafter N.M., *Adding wildcards to the Java programming language*. Journal of Object Technology 3 (2004) 97.
[7] Cook W.R., Hill W.L., Canning P.S., *Inheritance is not subtyping*. In: Proceedings of ACM Symposium on Principles of Programming Languages (POPL), ACM Press, (1990) 125.
[8] Leroy X., *The Objective Caml System, release 3.08. Documentation and user's manual*. INRIA. (2004) http://caml.inria.fr.
[9] Cardelli L., Wegner P., *On understandind types, data abstraction and polymorphism*. Computing Surveys 17 (1985) 471.
[10] Cardelli L., *Type systems*. In: Tucker A.B., ed.: Handbook of Computer Science and Engineering. CRC Press, (1997) 2208.
[11] Abadi M., Cardelli L., *A Theory of Objects,* Springer-Verlag, (1996).
[12] Bruce K.B., *Foundations of Object-Oriented Languages*. MIT Press, (2002).
[13] Castagna G., *Object-Oriented Programming. A Unified Foundation,* Birkhäuser, (1997).
[14] Pierce B.C., *Types and Programming Languages,* MIT Press, (2002).
[15] Torgersen M., Ernst E., Plesner Hansen C., *Wild FJ*. In: Proceedings of the Twelfth Workshop on Foundations of Object-Oriented Languages (FOOL), ACM Press, (2005).