Parametric Polymorphism for Java: A Reflective Approach

By Jose H. Solorzano and Suad Alagic

Presented by Matt Miller

February 20, 2003

Outline

- Motivation
- Key Contributions
- Background
  - Parametric Polymorphism
  - Java Core Reflection
- Survey of Approaches
- Comments
- Conclusions

Motivation

- Java's current "parametric polymorphism" is to make all the parameters a generic superclass (e.g., Object)
- This requires explicit downcasts at run-time when accessing objects. The downcast hinders performance and requires an extra burden on the programmer.
- Previous approaches do not consider the reflective properties of objects

Key Contributions

- Provides correct reflective solutions for implementing parametric polymorphism
- Develops compact representation of run-time class objects
- Proposes a technique for handling static variables with parametric polymorphism
- Gives overview and comparison of existing approaches

Parametric Polymorphism Categories

- Parametric Polymorphism
  - A generic class has formal type parameters rather than actual types
- Bounded
  - The formal type parameter is specified to have an upper bound other than Object
- F-Bounded
  - The upper bound is recursively specified
  - Useful when binary operations are used on data

Parametric Polymorphism Category Examples

Illustration 1.1 *(Generic Class)*

```
public class Collection<T>
{ ...
  public boolean exists (T aElement);
  public void    addElement (T aElement);
  public void    removeElement (T aElement);
}
```

Illustration 1.2 *(Bounded Type Quantification)*

```
public interface Ord
{ ...
  public boolean lessThan (Object aObj);
}

public class OrdCollection<T implements Ord>
extends Collection<T>
{ ...
}
```

Illustration 1.3 *(F-bounded Polymorphism)*

```
public interface Ordered<T>
{ ...
  public boolean lessThan (T aObject);
}

public class OrderedCollection<T implements Ordered<T>>
extends Collection<T>
{ ...
}
```

Java Core Reflection (JCR)

- Applications can acquire run-time information about the class of an object
- Allows discovery of methods which can then be invoked
- Complete reflection should allow run-time queries for the generic classes and classes instantiated from generics

Evaluation of Approaches

- Is the source code of generic classes required during compiling
- How much memory do class objects use
- How much indirection is necessary to access methods
- What reflective information is available
- How are static variables handled

Approach 1: Textual Substitution (TS)

- Similar to C++ templates
- Requires source code of generic class at compile-time for instantiated classes. Does a macro expansion.
- Complete type-checking is only done when an instantiation of the generic class is encountered
- Allows flexibility because classes do not have to explicitly declare the implementation of an interface

Approach 2: Homogenous Translation (HM)

- Compiler translates instantiations to upper bound. Thus, run-time checks guaranteed to be correct.
- Only one class file and object per generic
- Only requires compiler changes
- Reflection is incorrect
  - Classes will be generics
  - Parameter types will be bounds
- Potential security hazard

```
class Pair<elem> {
    elem x;  elem y;
    Pair (elem x, elem y) {this.x = x; this.y = y;}
    void swap () {elem t = x; x = y; y = t;}
}

Pair<String> p = new Pair("world!", "Hello,");
p.swap();
System.out.println(p.x + p.y);
```

Original Code

```
class Pair {
    Object x;  Object y;
    Pair (Object x, Object y) {this.x = x; this.y = y;}
    void swap () {Object t = x; x = y; y = t;}
}
Pair p = new Pair((Object)"world!", (Object)"Hello,");
p.swap();
System.out.println((String)p.x + (String)p.y);
```

Compiler Translation

HM Security Hazard

```
interface Channel {…}
class Collection<T implements Channel> {
    … add(T anElement); …
}
class SecureChannel implements Channel {…}
class InsecureChannel implements Channel {…}
…
Collection<SecureChannel> c = new Collection<SecureChannel>;
persistentStore(“Collection1”, c);
…
Collection c2 = (Collection) persistentGet(“Collection1”);
    // add method takes type Channel
c2.add(new InsecureChannel());  // No errors
```

Approach 3: Heterogeneous Translation (HT)

- Separate class file and object created for each new instantiation
- Run-time info for instantiated classes is correct
- May produce many nearly identical classes
- No run-time information available for generic classes. They are never loaded.

```
class Pair<elem> {
    elem x;  elem y;
    Pair (elem x, elem y) {this.x = x; this.y = y;}
    void swap () {elem t = x; x = y; y = t;}
}

Pair<String> p = new Pair("world!", "Hello,");
p.swap();
System.out.println(p.x + p.y);
```

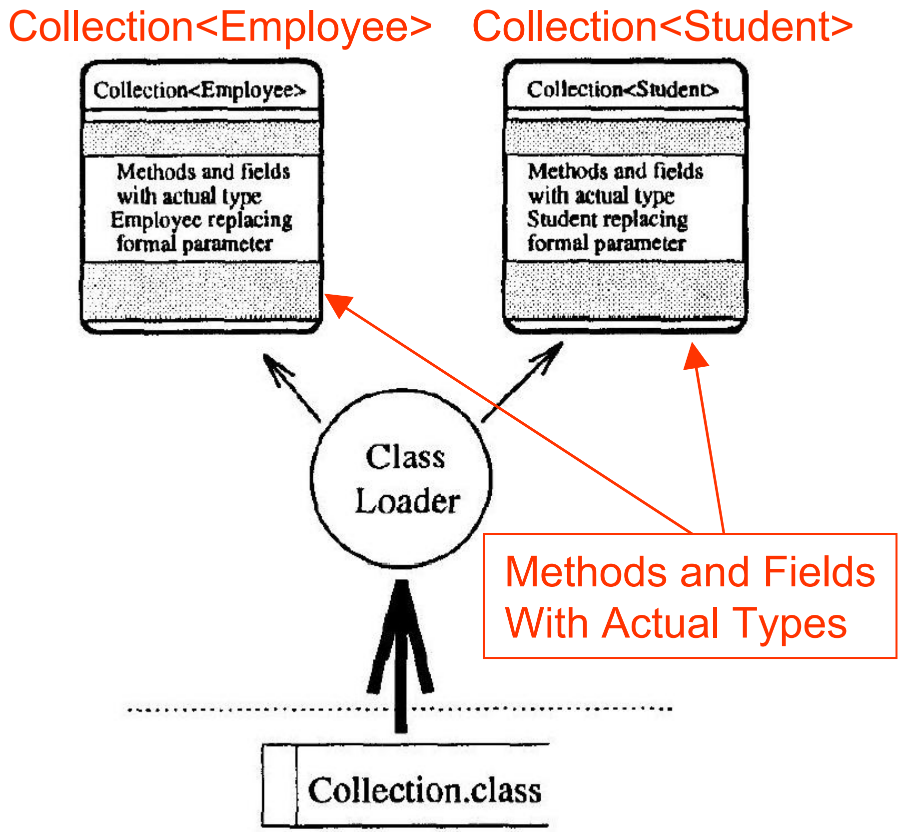
Original Code

```
class Pair_String {
    String x;  String y;
    Pair_String (String x, String y) {this.x = x; this.y = y;}
    void swap () {String t = x; x = y; y = t;}
}
Pair_String p = new Pair_String("world!", "Hello,");
p.swap(); System.out.println(p.x + p.y);
```

Compiler Translation

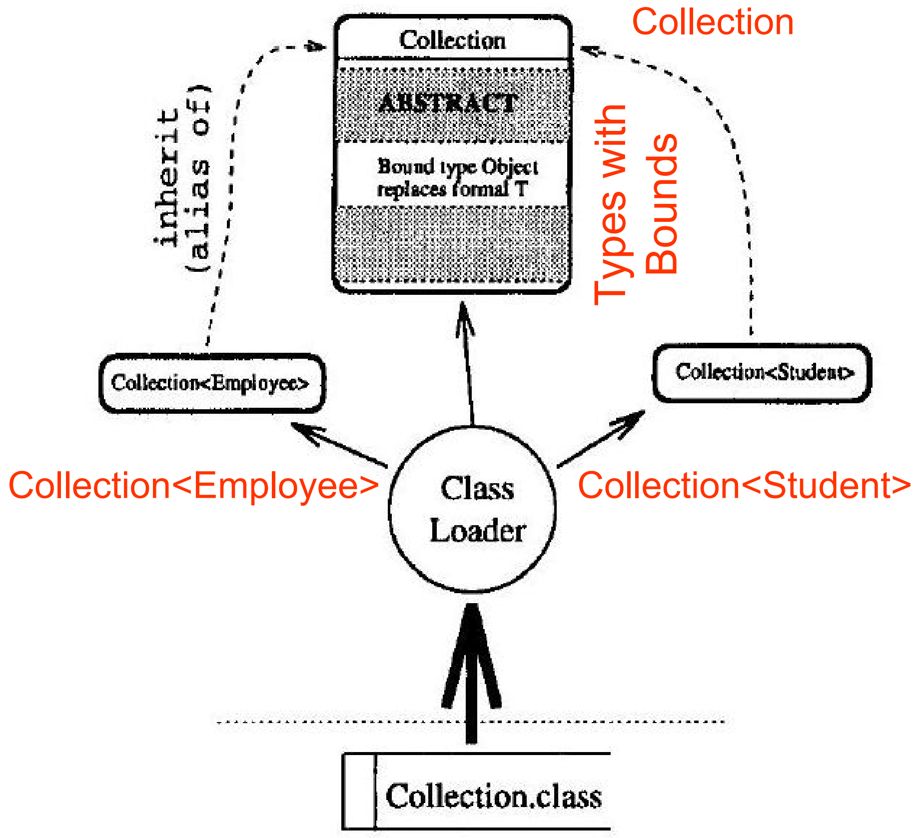
Approach 4: Load-Time Instantiation (LI)

- Extend class loader produce heterogeneous class objects from homogenous class file
- Improves HT by not producing redundant class files
- Same reflective capabilities as HT



Proposed Approach 1: Inheritance and Alias Classes (IH & AC)

- Similar to LI except instantiated classes are nearly empty and access code through generic class
- May require extra level of lookup for methods
- Parameters are reported as bound type
- Alias is a new relationship to correctly report the superclass of an object

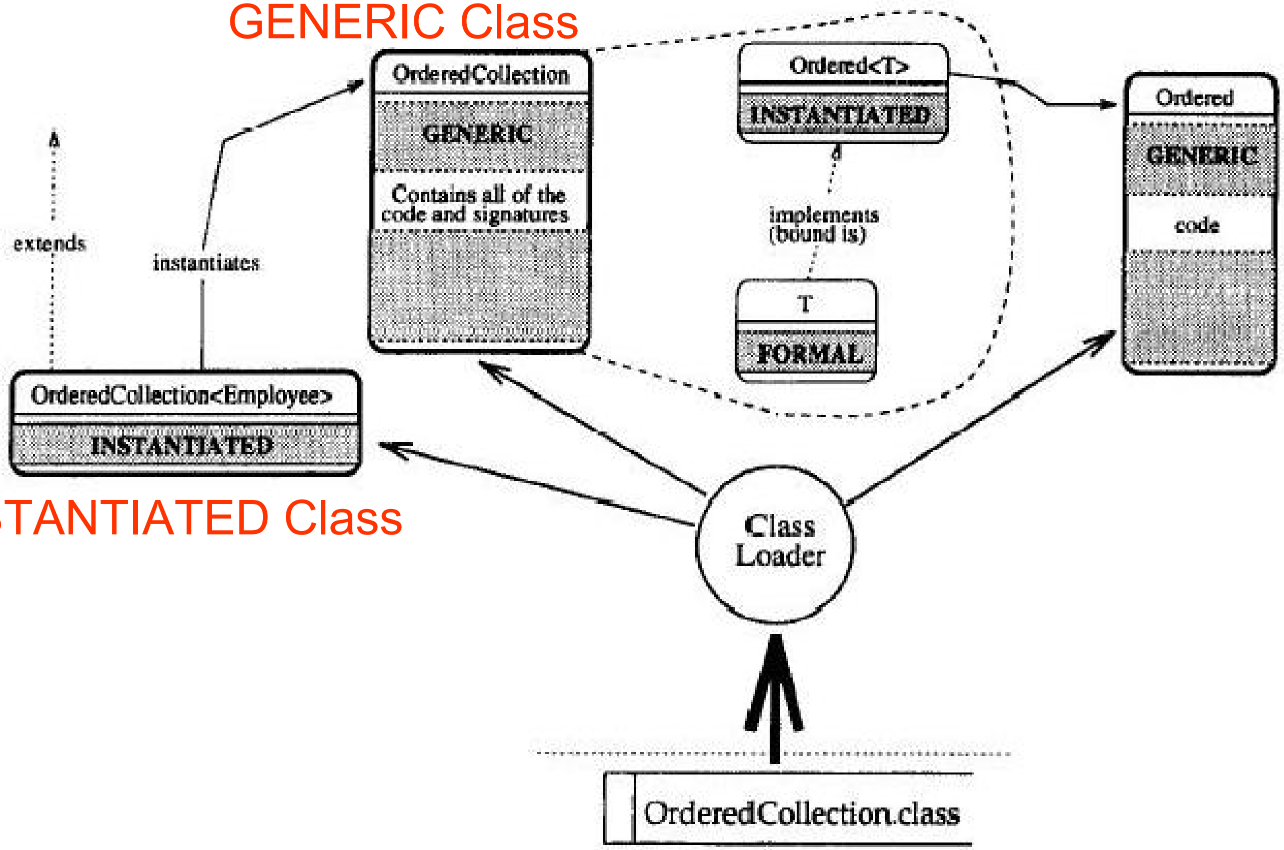


Proposed Approach 2: Extended Java Core Reflection (RF)

- Requires modifications to JVM, class loader and JCR classes
- Add class types GENERIC, INSTANTIATED and FORMAL
- Static variables can be stored in generic class or instantiated class
- Correct JCR available for each class

RF Illustration

GENERIC Class



INSTANTIATED Class

RF Changes to JCR

Illustration 8.1 *(Proposed Extensions to JCR)*

```
public class Class
{
  // Usual methods:
  public boolean isInterface();
  public boolean isAbstract();
  public String  getName();
  public Method  getDeclaredMethod (String aName,
                 Class[] aParamTypes);

  // RF methods for instantiated classes:
  public Class   getGeneric();
  public Class[] getActualParameters();

  // RF methods for formal type parameters:
  public Class   getUpperBound();
  public int     getPositionOfFormal();

  // RF methods for generic classes:
  public Class[] getFormalParameters();
         Class   instantiate (Class[] aArg);
}
```

Standard JCR Methods

INSTANTIATED Methods

FORMAL TYPE Methods

GENERIC Methods

Proposed Approach 3: Generic Code Sharing (RS)

- More efficient access to reflective information than RF
- No formal parameter classes
- Instantiated classes have actual method signatures which refer to the same generic code
- Reflection is less correct. Bound types are reported for generic classes instead of formal parameters.

Summary of Approaches

- HM is best for memory usage. IH/AC, RF and RS are better than HT/LI.
- IH/AC and RF require extra level of indirection from instantiated class to method and field signatures.
- Reflective Capabilities
  - HM is incorrect. Objects of different type instantiations cannot be dynamically distinguished. Multiple dispatch not possible.
  - HT/LI is more correct. Provide types of instantiated classes and correct parameter types for methods and fields.
  - RF is most correct. Gives actual types for instantiated classes, formal types for generics and bounds for formal parameters.
  - RS is slightly less correct. Generics only provide bound information, not formal type parameters.

Comments

- No performance evaluation of implementations
- Primitives still require extra overhead of wrapper classes
- Could lead to complex class hierarchy in large systems with many generic types

Conclusions

- Demonstrates how parametric polymorphism could be added to Java in a way that is compact and correct with respect to JCR
- Allows static variables per generic or per instantiation
- Surveys and compares existing approaches to the problem

Bonus Slides

Persistent Store

- Emerging technology for Java allows objects to outlive the current application
- All objects referenced within a stored object also become persistent. This includes an implicit reference to the Class object.
- Need reflection to type check when retrieving persistent object
- Should limit redundancy among instantiated classes

Persistent Store vs. Serialization

- Serialization: Creates a series of bytes to represent an object and all objects reachable from it
- Successive retrievals of a serialized object will have a different identity.
- Serialization suffers from “big inhale”. That is, one must wait for the entire byte stream to be loaded even if only a small portion of the data is needed.

Multiple Dispatch

- Single dispatch (e.g., Java) chooses the method based on the run-time type of caller and the static type of the input parameters
- Multiple dispatch would allow the choice of the method to also be a function of the input parameter run-time types

```
Public class Shape {
  public boolean intersect(Shape s) {
    /***/
  }
} /* End Class Shape */

Public class Rectangle extends Shape {
  public boolean intersect(Rectangle r) {
    /***/
  }
} /* End Class Rectangle */
```

```
Rectangle r1, r2;
Shape s1, s2;
boolean b1, b2, b3, b4;
r1 = new Rectangle( /***/ );
r2 = new Rectangle ( /***/ );
s1 = r1;
s2 = r2;
B1 = r1.intersect(r2);    Rectangle.intersect()
B2 = r1.intersect(s2);    Shape.intersect()
B3 = s1.intersect(r2);    Shape.intersect()
B4 = s1.intersect(s2);    Shape.intersect()
```

- Consider s1.intersect(s2) whose static types are Shape.
- If at run time both s1 and s2 are of type Circle, then the first and third of these methods are applicable along with Shape's default "intersect" method.
- The third one is most specific so it is executed

```
Public class Circle extends Shape {
  public boolean intersection(Shape s) {
    /* Code for a circle against a shape */
  }

  public boolean intersect(Shape@Rectangle r) {
    /* Efficient code against a Rectangle */
  }

  public boolean intersect(Shape@Circle c) {
    /* Efficient code against a Circle */
  }
}
```

Table 8.1

Abbrev.	Description	Source	Memory	Perform.	Actuals	Mult. disp.	Reflection
TS	Textual Substitution			++	+	+	++
HM	Homogeneous	+	+++	++			
HT	Heterogeneous	+		++	+	+	++
LI	Load-time instantiation	+		++	+	+	++
IH	Inst. by inheritance	+	++	+	+		+
AC	Inst. by class aliasing	+	++	+	+		+
RF	Reflect. technique 1	+	++	+	++	+	++++
RS	Reflect. technique 2	+	++	++	++	+	+++

Table 8.1: Evaluation of implementation techniques for parametric polymorphism.

Detailed RF Changes to JCR

Illustration 8.1 *(Proposed Extensions to JCR)*

```
public class Class
{
  // Usual methods:
  public boolean isInterface();
  public boolean isAbstract();
  public String  getName();
  public Method  getDeclaredMethod (String aName,
                 Class[] aParamTypes);
  ...
  // RF general methods:
  public boolean isGeneric();
  public boolean isInstantiated();
  public boolean isFormalParameter();

  // RF methods for instantiated classes:
  public Class   getGeneric();
  public Class[] getActualParameters();

  // RF methods for formal type parameters:
  public Class   getUpperBound();
  public int     getPositionOfFormal();

  // RF methods for generic classes:
  public Class[] getFormalParameters();
         Class   instantiate (Class[] aArg);
}
```

Illustration 8.2 *(Extension to class* `Method`*)*

```
public class Method
{
  // Usual methods:
  public String getName();
  public Class  getReturnType();
  public Object invoke (Object aRecv, Class[] aParams);
  // [...] New RF method:
  public Method getActualSignature (Class[] aAct);
}
```

Illustration 8.3 *(Extension to a method of* `Class`*)*

```
public class Class
{ ...
  public Method getDeclaredMethod (String aName,
                Class[] aParamTypes)
  {
    if (isInstantiated())
    { // Assume single dispatch
      Method pMethod = getGeneric().
             getDeclaredMethod(aName, aParameters);
      return pMethod.getActualSignature (
             getActualParameters());
    }
    else
    { ... // Other cases
    }
  }
}
```

Issues with Parametric Polymorphism in Java

- Static Fields
- Explicit interface implementation versus equivalent class structure
- Constructors of subtypes may differ from those of the supertype
- Duplicate methods after instantiation
- Subtyping semantics

Subtype Constructor Problem

- Subtype of person may have no constructor which matches signature
- Or, subtype may match either of the signatures

Illustration 5.1 *(Constructors of Formal Type Parameters)*

```
public class PersonCollection<T extends Person>
{ ...
  public void a_method()
  {
    // Which of these statements is incorrect?
    T person1 = new T(``Jones'');
    T person2 = new T(``Smith'', 50000);
  }
}
```

Duplicate Method Problem

```
class Collection<T> {
    boolean add(T element);
    boolean add(Employee element);
}

Collection<Employee> c;
```

Subtyping Semantic Problem

```
class Collection<A> {…}
class Y extends X {…}
Collection<Y> y = new Collection<Y>;
Collection<X> x = y;  // Compile time error
x.insert(new X());  // Type violation
```

But, this is legal in Java:

```
Y[ ] y = new Y[10];
X[ ] x = y;
x[0] = new X();
```