

# Type Erasure: Breaking the Java Type System

Suad Alagić, Brian Cabana, Jeff Faulkner and  
Mark Royer

Department of Computer Science  
University of Southern Maine

Boston University, Fall 2004

# Controversy

- Years of research for correct solution
- Solutions based on a wrong idiom
- J2SE DK 5.0 based on a wrong solution
- JDK 5.0: Violations of the type system
- A correct solution in fact exists

# Previous Work

- Pizza and GJ
- PolyJ
- Heterogeneous Solutions
- Reflective Solution
- NextGen and MixGen
- C#
- Java Specification Request – JSR14

# Bounded Parametric Polymorphism

```
class SortedList<T extends Comparable> {  
    private T [] elements;  
    public SortedList<T>() {  
        elements = new T[size];  
    }  
    public int add(T t)  
    public T remove(int index)  
    public int size();  
}
```

# Type Erasure

```
class SortedList {  
    private Comparable [] elements;  
    public SortedList() {  
        elements = new Comparable[size];  
    }  
    public int add(Comparable t);  
    public Comparable remove(int i);  
    public int size();  
}
```

# Implications of Type Erasure

- Consider `SortedList<Student>` and `SortedList<Book>`
- Clearly different types but
- One and the same class object `SortedList<Comparable>`
- Incorrect run-time type information

# Persistence Systems Implications

- Make `SortedList<Employee> employees` persistent
- Reachability: the class object becomes persistent
- But `employees.getClass()` is `SortedList<Comparable>`
- Is there an object in the database of type `SortedList<Employee>`?

# J2SE DK 5.0

## Unforgivable Pitfalls:

- Violation of the Java type system
- Incorrect run-time type information
- Java Core Reflection implications
- Overloading
- Serialization and type casts

## Reasons for failure:

- Generic Idiom is provably wrong
- Java Virtual Machine must be extended

# Violation of Type System

```
Vector<Integer> v = new Vector<Integer>();  
myMethod(v);  
Integer i = v.lastElement();
```

```
public static void myMethod(Object collection)  
{  
    // Vector<Integer> instanceof and cast not allowed  
    if (collection instanceof Vector)  
        ((Vector)collection).add("trouble");  
}  
}
```

# Subtyping Rules Violated

```
Vector<Integer> a = new Vector<Integer>();
```

```
Vector b = new Vector();
```

```
b.add(new Object());
```

```
a = b;
```

```
...
```

```
Integer i = a.remove(0);
```

# Problems with Overloading

```
public class MyClass {  
    public void myMethod(Vector<Integer> list) {}  
    public void myMethod(Vector<String> list) {}  
    // Both methods arguments appear as type Vector  
}
```

```
public class LinkedList <G extends Comparable> {  
    public void myMethod(G g) {}  
    public void myMethod(Comparable c) {}  
    // Both method arguments appear as type Comparable  
}
```

# Problems with Java Core Reflection

```
public class SomeClass {  
    public void myMethod(Vector<String> value) {  
        String s = value.lastElement();  
    }  
}
```

```
public class MyClass {  
    public static void main(String[] args) {  
        SomeClass c = new SomeClass();  
        Vector<Integer> v = new Vector<Integer>();  
        Method m = c.getClass().getMethod("myMethod",  
            v.getClass());  
        v.add(new Integer(123));  
        m.invoke(c, v);  
    }  
}
```

# Run-time Exceptions

- Covariant array subtyping:  
Employee [] subtype of Person[]
- Static type checking rules violated
- Dynamic type checks work
- GJ: wrong run-time type info, no standard exceptions
- Failures at unpredictable places

# Implementing Interfaces

```
interface A{  
    Vector<Integer> m(Vector<Integer> x); }
```

```
interface B extends interface A {  
    Vector<String> m(Vector<String> x);}
```

```
Class C implements B {  
    Vector<String> m(Vector<String> x) {return null;}  
    // Vector<Integer> m(Vector<Integer> x) {return null;}  
}
```

# Type Names

```
class Ordered<T extends Comparable>{  
    boolean lessThan(T x) {...}  
}
```

```
Ordered<Integer> I= new Ordered<Integer>;  
Ordered<String> S = new Ordered<String>;
```

```
I.getClass().getName() == S.getClass().getName()  
                        == "Ordered"
```

```
I = (Ordered)S;  
S= (Ordered)I;
```

# Serialization

```
Class WriteObject {
```

```
...
```

```
    FileOutputStream fileout = new  
        FileOutputStream("myObject.obj");
```

```
    ObjectOutputStream out = new  
        ObjectOutputStream(fileout);
```

```
    Vector<Integer> s = new Vector<Integer>();
```

```
    s.add(new Integer(5));
```

```
    out.writeObject(s);
```

```
}
```

# Serialization and Type Casting

```
class ReadObject {  
    ...  
    FileInputStream filein = new  
        FileInputStream("myObject.obj");  
    ObjectInputStream in = new ObjectInputStream(filein);  
    Vector<String> s = null;  
    try {  
        s = (Vector<String>) in.readObject();  
        // Warning even for Vector<Integer>  
    } catch (InvalidClassException e) {...}  
  
    System.out.println("Read " + s.get(0));  
    // Cast exception on s.get(0)  
}
```

# Components of Correct Solution

- Java Class File Structure
  - Optional Attributes
  - Type Descriptors
- Extended Class Loader
- JVM Modifications
  - Class Objects
  - Class Loading
  - Extended Java Core Reflection

# Class File Representation

- Class file for `SortedList<T extends Comparable>`
- Flexibility in JVM: optional attributes
- Optional class attributes: positions, names and bounds of type parameters
- Distinction between type parameters and bound types
- Likewise for methods and fields

# Parametric Class Attribute

- Type parameter information
  - Both for bound and actual type
- Field descriptors
- Method descriptors
- Constant pool instructions

# Type Descriptors

- T[] elements: [Ljava/lang/Comparable;
- int add(T x): (Ljava/lang/Comparable;)I
- Class object for SortedList<Integer>:  
[Ljava/lang/Integer; and  
(Ljava/lang/Integer;)I
- Loader action

# Constant Pool: SortedList<T extends Comparable>

Index    Type    Entry

3	Class	27
4	Class	28
5	Class	29



Actual Type

27	UTF8	java/lang/Comparable
28	UTF8	java/lang/Object
29	UTF8	SortedList

35	UTF8	T
36	Class	27
37	UTF8	SortedList\$\$#0\$\$
38	UTF8	#0



Bound Type

# Constant Pool: SortedList<Integer>

Index	Type	Entry
3	Class	38
4	Class	28
5	Class	37

← Actual Type

27	UTF8	java/lang/Comparable
28	UTF8	java/lang/Object
29	UTF8	SortedList

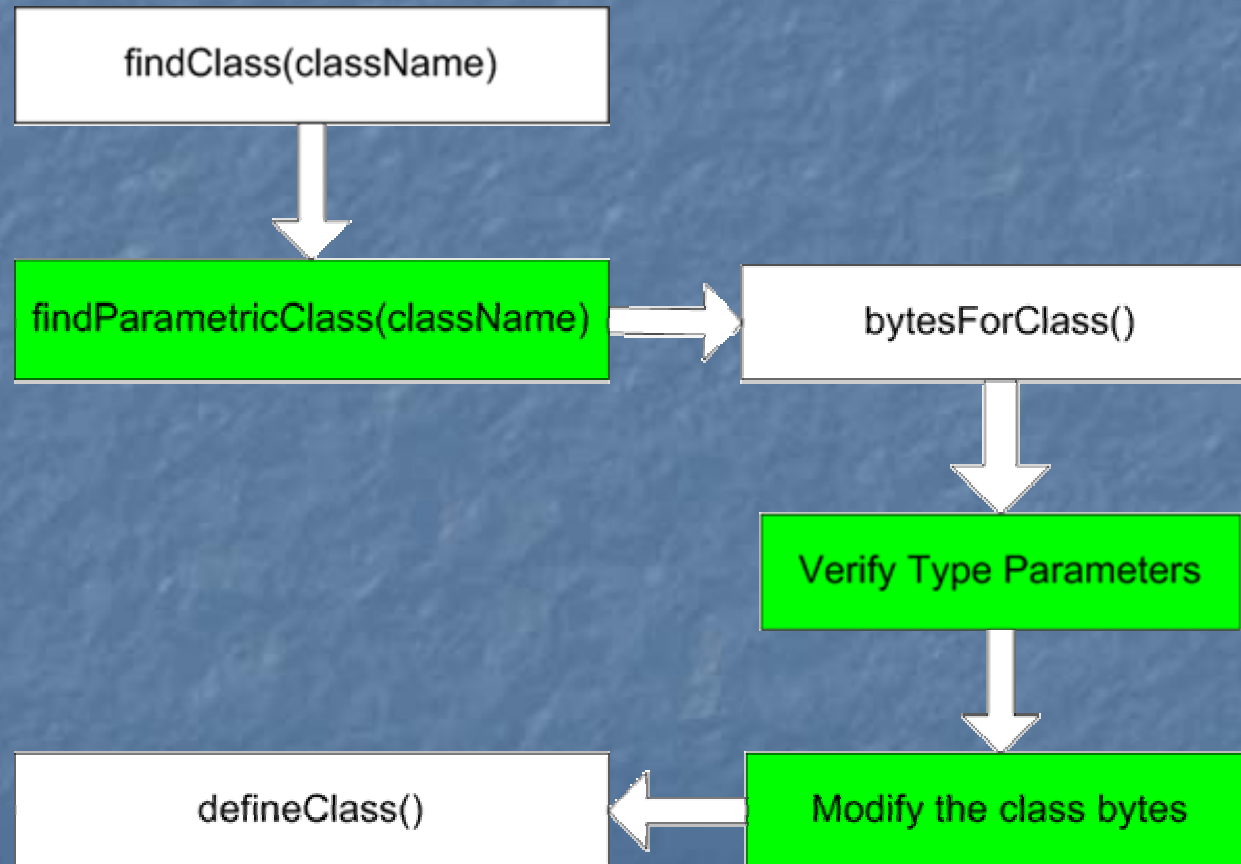
35	UTF8	T
36	Class	27
37	UTF8	SortedList\$\$java/lang/Integer\$\$
38	UTF8	java/lang/Integer

← Bound Type

# Loading Class Objects

- Creates class objects with correct actual type information
- Class methods and fields have correct descriptors
- Class loading model preserved
- Extended ClassLoader problems
- URLClassLoader and native class loader must be extended

# Class Loading



# Java Core Reflection

- Extending final classes:  
Class, Method and Field
- Additional methods
- Is a class parametric, type parameters, bound types, actual type parameters
- No negative effect on JCR
- Recompile of the whole platform

# Code Multiplicity

- C# solution avoids code multiplicity:
  - multiple v-tables with actual types
- NextGen uses multiple class objects:
  - elaborate representation
- JVM requires more serious changes
- Is this justified?
- Type safety versus some memory penalty

# Conclusions

- JDK 5.0 is based on a wrong idiom (GJ type erasure)
- No correct solution unless JVM is extended
- It is possible to construct an extension which does not violate integrity of JVM
- Legacy code will not be affected

# Conclusions

- Java is not type safe any more!
- There is a better way!