Choosing Efficient Inheritance Patterns for Java Generics

Fernando Trinciante, Isaac Sjoblom, Elena Machkasova

University of Minnesota, Morris

Midwest Instruction and Computing Symposium
Eau Claire, WI, 2010
Java Compilation Model

- **Two stages system.**
  - Static compilation into bytecode
  - Dynamic processing
    - Interpretation.
    - Optimization.
    - Compilation to native code.

- **Java Virtual Machine (JVM)**
  - Oracle’s HotSpot JVM

- **Just-In-Time Compilers (JIT).**
  - Bycode interpretation.
  - Program analysis and profiling.
  - Compilation to native code.
Java Compilation Model

- Two stages system.
  - Static compilation into bytecode
    - Dynamic processing
      - Interpretation.
      - Optimization.
      - Compilation to native code.
  - Java Virtual Machine (JVM)
    - Oracle’s HotSpot JVM
  - Just-In-Time Compilers (JIT).
    - Bycode interpretation.
    - Program analysis and profiling.
    - Compilation to native code.
Java Compilation Model

- Two stages system.
  - Static compilation into bytecode
  - Dynamic processing
    - Interpretation.
    - Optimization.
    - Compilation to native code.

- Java Virtual Machine (JVM)
  - Oracle’s HotSpot JVM

- Just-In-Time Compilers (JIT).
  - Bycode interpretation.
  - Program analysis and profiling.
  - Compilation to native code.
Java Compilation Model

- Two stages system.
  - Static compilation into bytecode
  - Dynamic processing
    - Interpretation.
    - Optimization.
    - Compilation to native code.

- Java Virtual Machine (JVM)
  - Oracle’s HotSpot JVM

- Just-In-Time Compilers (JIT).
  - Bycode interpretation.
  - Program analysis and profiling.
  - Compilation to native code.
Java Compilation Model

- Two stages system.
  - Static compilation into bytecode
  - Dynamic processing
    - Interpretation.
    - Optimization.
    - Compilation to native code.

- Java Virtual Machine (JVM)
  - Oracle’s HotSpot JVM

- Just-In-Time Compilers (JIT).
  - Bycode interpretation.
  - Program analysis and profiling.
  - Compilation to native code.
Overview of Java Bytecode

- Portable and platform independent code.
- Types of instructions: $i = \text{integer}$, $a = \text{reference to object}$.

<table>
<thead>
<tr>
<th>Operations</th>
<th>Instructions (examples)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load and store</td>
<td>aload, istore</td>
</tr>
<tr>
<td>Typecasting</td>
<td>checkcast</td>
</tr>
<tr>
<td>Method call (public)</td>
<td>invokevirtual</td>
</tr>
<tr>
<td>Method call (private)</td>
<td>invokespecial</td>
</tr>
<tr>
<td>Method call (interface)</td>
<td>invokeinterface</td>
</tr>
<tr>
<td>Method returns</td>
<td>areturn, ireturn</td>
</tr>
</tbody>
</table>
**Overview of Java Bytecode**

- Portable and platform independent code.
- Types of instructions: \( i = \text{integer}, a = \text{reference to object} \).

<table>
<thead>
<tr>
<th>Operations</th>
<th>Instructions (examples)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load and store</td>
<td>aload, istore</td>
</tr>
<tr>
<td>Typecasting</td>
<td>checkcast</td>
</tr>
<tr>
<td>Method call (public)</td>
<td>invokevirtual</td>
</tr>
<tr>
<td>Method call (private)</td>
<td>invokespecial</td>
</tr>
<tr>
<td>Method call (interface)</td>
<td>invokeinterface</td>
</tr>
<tr>
<td>Method returns</td>
<td>areturn, ireturn</td>
</tr>
</tbody>
</table>
Overview of Java Bytecode

- Portable and platform independent code.
- Types of instructions: \(i = \text{integer}, a = \text{reference to object}\).

<table>
<thead>
<tr>
<th>Operations</th>
<th>Instructions (examples)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load and store</td>
<td>aload, istore</td>
</tr>
<tr>
<td>Typecasting</td>
<td>checkcast</td>
</tr>
<tr>
<td>Method call (public)</td>
<td>invokevirtual</td>
</tr>
<tr>
<td>Method call (private)</td>
<td>invokespecial</td>
</tr>
<tr>
<td>Method call (interface)</td>
<td>invokeinterface</td>
</tr>
<tr>
<td>Method returns</td>
<td>areturn, ireturn</td>
</tr>
</tbody>
</table>
Overview of Java Bytecode

- Portable and platform independent code.
- Types of instructions: $i =$ integer, $a =$ reference to object.

<table>
<thead>
<tr>
<th>Operations</th>
<th>Instructions (examples)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load and store</td>
<td>aload, istore</td>
</tr>
<tr>
<td>Typecasting</td>
<td>checkcast</td>
</tr>
<tr>
<td>Method call (public)</td>
<td>invokevirtual</td>
</tr>
<tr>
<td>Method call (private)</td>
<td>invokespecial</td>
</tr>
<tr>
<td>Method call (interface)</td>
<td>invokeinterface</td>
</tr>
<tr>
<td>Method returns</td>
<td>aretturn, ireturn</td>
</tr>
</tbody>
</table>
Client and Server Modes of HotSpot

- **Client:**
  - Fast program startup.
  - Minimal inlining.
  - Compilation threshold \(\sim 1500\).

- **Server:**
  - Tuned for server-side application.
  - Deep inlining.
  - Compilation threshold \(\sim 10000\)
Client and Server Modes of HotSpot

Client:
- Fast program startup.
- Minimal inlining.
- Compilation threshold $\sim 1500$.

Server:
- Tuned for server-side application.
- Deep inlining.
- Compilation threshold $\sim 10000$.
Client and Server Modes of HotSpot

Client:
- Fast program startup.
- Minimal inlining.
- Compilation threshold \( \sim 1500 \).

Server:
- Tuned for server-side application.
- Deep inlining.
- Compilation threshold \( \sim 10000 \).
Client and Server Modes of HotSpot

Client:
- Fast program startup.
- Minimal inlining.
- Compilation threshold $\sim 1500$.

Server:
- Tuned for server-side application.
- Deep inlining.
- Compilation threshold $\sim 10000$
Java Generic Types allow a data container to hold several different types of elements.

**Data container ArrayList declaration and instantiation**

```java
public class ArrayList<T> implements List {
    //ArrayList Constructor and Methods
}

ArrayList<String> strings = new ArrayList<String>();
ArrayList<Integer> integers = new ArrayList<Integer>();
strings.add("hello");
String hello = strings.get(0);
```
Type Bounds limit generic type parameters.

**Comparable Type Bound**

```java
public class MyComparableList<T extends Comparable>{
    //methods and constructor
}
```
In Java only one definition of a type with default bound `Object` or a specified bound, (e.g. `Comparable`) is compiled.

Instances of `ArrayList` such as `ArrayList<String>` and `ArrayList<Integer>` reference the same definition.

Example of `ArrayList` with String Cast

```java
public class BytecodeExample {
    public static void main(String[] args) {
        ArrayList<String> alString = new ArrayList<String>();
        //add some elements to alString
        String exampleString = alString.get(3);
    }
}
```

```
38 invokevirtual #30 <java/util/ArrayList.get>
41 checkcast #34 <java/lang/String>
44 astore_2
```
Narrowing the Type Bound:

- When a Java generic type inherits from a generic interface or class, the supertype may have a less restrictive type bound than the subtype itself.

List Interface Declaration

```java
interface List<T extends Object>
```

Narrowed Type Bound

```java
class NArrayList<T extends Number> implements List<T>
```
Figure: ArrayList hierarchy in Java Collections Library.
ListReader is our own class, and is not part of the JCF.
ListReader is generic.

**Example of ListReader**

```java
ListReader<Integer> reader = new ListReader<Integer>();
```

- Testing of ArrayList.
- Repeated test with large loops (for instance 400,000,000).
- Calls the method (e.g. `get`) via a List interface variable (`invokeinterface`).

**Example of ListReader: testing get method**

```java
reader.testGet(theList, numLoops);
```
ListReader is our own class, and is not part of the JCF.
ListReader is generic.

Example of ListReader

```java
ListReader<Integer> reader = new ListReader<Integer>();
```

- Testing of ArrayList.
  - Repeated test with large loops (for instance 400,000,000).
  - Calls the method (e.g. `get`) via a List interface variable (`invokeinterface`).

Example of ListReader: testing `get` method

```java
reader.testGet(theList, numLoops);
```
ListReader is our own class, and is not part of the JCF.
ListReader is generic.

Example of ListReader

```java
ListReader<Integer> reader = new ListReader<Integer>();
```

- Testing of ArrayList.
- Repeated test with large loops (for instance 400,000,000).
  - Calls the method (e.g. `get`) via a List interface variable (`invokeinterface`).

Example of ListReader: testing `get` method

```java
reader.testGet(theList, numLoops);
```
ListReader

- ListReader is our own class, and is not part of the JCF.
- ListReader is generic.

Example of ListReader

ListReader<Integer> reader = new ListReader<Integer>();

- Testing of ArrayList.
- Repeated test with large loops (for instance 400,000,000).
- Calls the method (e.g. get) via a List interface variable (invokeinterface).

Example of ListReader: testing get method

reader.testGet(theList, numLoops);
Figure: ArrayList hierarchy in JCF: narrowing.
Eight Versions of Code

- **O, OO, and S:** The O-group.
- **AL, ALO, and ALS:** The AL-group bound narrowed.
- **LS and C:** The C-group.

<table>
<thead>
<tr>
<th>Name</th>
<th>Bounds</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td><code>ArrayList</code>: <code>Object</code>, <code>List</code>: <code>Object</code>, <code>ListReader</code>: <code>Comparable</code></td>
</tr>
<tr>
<td>OO</td>
<td><code>ArrayList</code>: <code>Object</code>, <code>List</code>: <code>Object</code>, <code>ListReader</code>: <code>Object</code></td>
</tr>
<tr>
<td>S</td>
<td><code>ArrayList</code>: <code>Object</code>, <code>List</code>: <code>Object</code>, <code>ListReader</code>: <code>Integer</code></td>
</tr>
<tr>
<td>AL</td>
<td><code>ArrayList</code>: <code>Integer</code>, <code>List</code>: <code>Object</code>, <code>ListReader</code>: <code>Comparable</code></td>
</tr>
<tr>
<td>ALO</td>
<td><code>ArrayList</code>: <code>Integer</code>, <code>List</code>: <code>Object</code>, <code>ListReader</code>: <code>Object</code></td>
</tr>
<tr>
<td>ALS</td>
<td><code>ArrayList</code>: <code>Integer</code>, <code>List</code>: <code>Object</code>, <code>ListReader</code>: <code>Integer</code></td>
</tr>
<tr>
<td>C</td>
<td>All hierarchy: <code>Integer</code></td>
</tr>
</tbody>
</table>
1. Check the index.
2. Return element from array.

**Method get in ArrayList**

```java
public T get(int index) {
    RangeCheck(index);
    return elementData[index];
}
```

**Method RangeCheck in ArrayList**

```java
private void RangeCheck(int index) {
    if (index >= size){
        throw new IndexOutOfBoundsException(
            "Index: " + index + ", Size: " + size);
    }
}
```
Method `get`

1. Check the index.
2. Return element from array.

```java
public T get(int index) {
    RangeCheck(index);
    return elementData[index];
}
```

```java
private void RangeCheck(int index) {
    if (index >= size){
        throw new IndexOutOfBoundsException(
            "Index: " + index + ", Size: " + size);
    }
}
```
public T get(int index) {
    RangeCheck(index);
    return elementData[index];
}

//Load ArrayList:
0  aload_0
1  iload_1
2  invokespecial #36 <researchutilArrayList.RangeCheck>
5  aload_0
//Get the elementData array:
6  getfield #12 <researchutilArrayList.elementData>
9  iload_1
//Load a reference to an element in the array:
10  aaload
//Return the reference to the calling method:
11  areturn
Testing Methodology

- Test runs on Client and Server modes.
  - Pure interpreted: JVM flag `-Xint`.
  - No-Inline: JVM flag `-XX:-Inline`
  - Regular: No JVM flag.
Figure:
Testing `get` method: no inlining mode

Figure:
Testing \textit{get method: regular mode}

Server: get 400,000,000; regular mode, 1.16.16 JVM

Figure:
Method \texttt{getEqual}

- Return boolean.

**Example: method \texttt{getEqual} on ArrayList**

```java
public boolean getEqual(int index) {
    RangeCheck(index);
    return (elementData[index] == elementData[(index+1) % size]);
}
```

65  \texttt{aaload}
66  \texttt{if_acmpne} 73 (+7)
69  \texttt{iconst\_1}
70  \texttt{goto} 74 (+4)
73  \texttt{iconst\_0}
74  \texttt{ireturn}

Testing `get_equal` method: interpreted mode

Server: `get-equal` 50,000,000; interpreted mode, no JIT

CPU Time (sec.)

Figure:
Summary of Observations

- The slowdown in AL-group (bound-narrowing).
- Slowdown: passing reference to an object.
- No slowdown: when passing primitive types.
- Likely explanation: a typecheck is performed.
- JIT optimizations remove the slowdown for `get`.
- Slowdown still exists: `add` and `set`.
- Complicated results; different for client and server.
Conclusions and Future Work

We discovered: a slowdown associated with bound narrowing.

JIT compensates for the slowdown for `get`.

Bound narrowing can be used in software development.

Future work:

- Find a clearer evidence of a typecheck (a ClassCastException thrown?)
- Explain behavior of other methods (`add`, `set`)
- Continue trying other JVMs (Jikes RVM, results not shown).
Selected References


Our Test Machine:
AMD AthlonTM 64 Processor 3200+, 512MB DDR RAM
Fedora Core 7, Java Version: Sun JDK 1.6.16
Time Binary: GNU time 1.7