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

Outline

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 = integer, a = reference to object.

Operations	Instructions (examples)
Load and store	aload, istore
Typecasting	checkcast
Method call (public)	invokevirtual
Method call (private)	invokespecial
Method call (interface)	invokeinterface
Method returns	areturn, ireturn

Overview of Java Bytecode

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

Operations	Instructions (examples)
Load and store	aload, istore
Typecasting	checkcast
Method call (public)	invokevirtual
Method call (private)	invokespecial
Method call (interface)	invokeinterface
Method returns	areturn, ireturn

Overview of Java Bytecode

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

Operations	Instructions (examples)
Load and store	aload, istore
Typecasting	checkcast
Method call (public)	invokevirtual
Method call (private)	invokespecial
Method call (interface)	invokeinterface
Method returns	areturn, ireturn

Overview of Java Bytecode

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

Operations	Instructions (examples)
Load and store	aload, istore
Typecasting	checkcast
Method call (public)	invokevirtual
Method call (private)	invokespecial
Method call (interface)	invokeinterface
Method returns	areturn, ireturn

Client and Server Modes of HotSpot

- Client:
  - Fast program startup.
  - Minimal inlining.
  - Compilation threshold ∼1500.
- Server:
  - Tuned for server-side application.
  - Deep inlining.
  - Compilation threshold ∼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 ∼1500.
- Server:
    - Tuned for server-side application.
    - Deep inlining.
    - Compilation threshold ∼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

Overview of Generic Types

Java Generic Types allow a data container to hold several different types of elements.

Data container `ArrayList` declaration and instantiation

```
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 Bound

Type Bounds limit generic type parameters.

Comparable Type Bound

```
public class MyComparableList<T extends Comparable>{
//methods and constructor
}
```

Type Erasure

- 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

```
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 Type Bound

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.

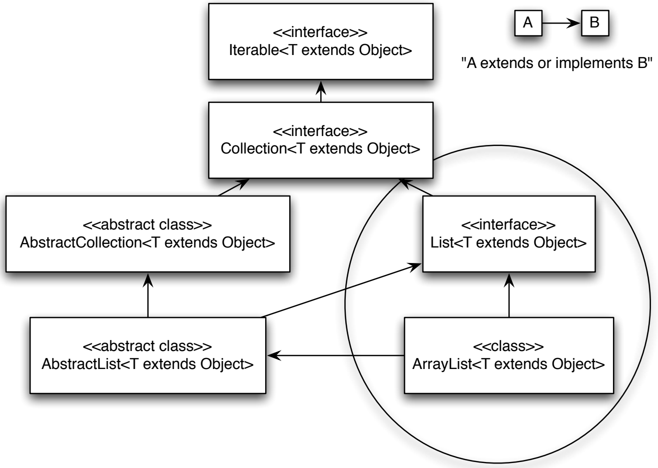
List Interface Declaration

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

Narrowed Type Bound

```
class NArrayList<T extends Number> implements List<T>
```

ArrayList Hierarchy



<<interface>>
Iterable<T extends Object>
A
B
"A extends or implements B"
<<interface>>
Collection<T extends Object>
<<abstract class>>
AbstractCollection<T extends Object>
<<interface>>
List<T extends Object>
<<abstract class>>
AbstractList<T extends Object>
<<class>>
ArrayList<T extends Object>

Figure: ArrayList hierarchy in Java Collections Library.

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);
```

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);
```

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);
```

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);
```

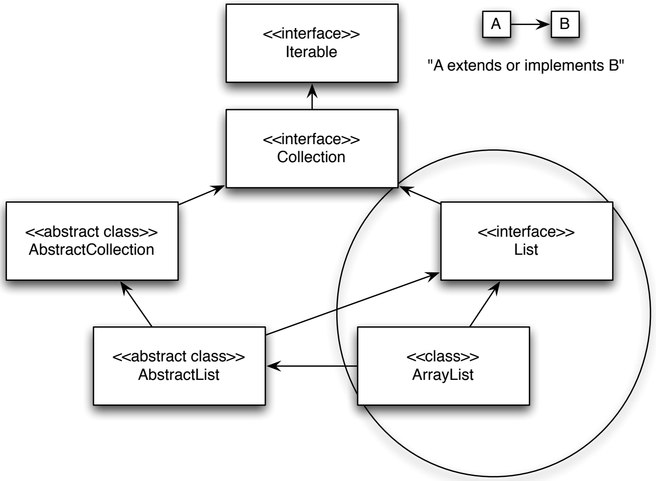


Figure: `ArrayList` hierarchy in JCF: narrowing.

Eight Versions of Code

- O, OO, and S: The O-group.
- AL, ALO, and ALS: The AL-group bound narrowed.
  ArrayList: Integer, List: Object.
- LS and C: The C-group.

Name	Bounds
O	ArrayList: Object, List: Object, ListReader: Comparable
OO	ArrayList: Object, List: Object, ListReader: Object
S	ArrayList: Object, List: Object, ListReader: Integer
AL	ArrayList: Integer, List: Object, ListReader: Comparable
ALO	ArrayList: Integer, List: Object, ListReader: Object
ALS	ArrayList: Integer, List: Object, ListReader: Integer
LS	ArrayList: Integer, List: Integer, ListReader: Integer
C	All hierarchy: Integer

Method get

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

Method get in ArrayList

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

Method RangeCheck in ArrayList

```
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.

Method `get` in `ArrayList`

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

Method `RangeCheck` in `ArrayList`

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

Method get bytecode

```
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.

Testing `get` method: interpreted mode

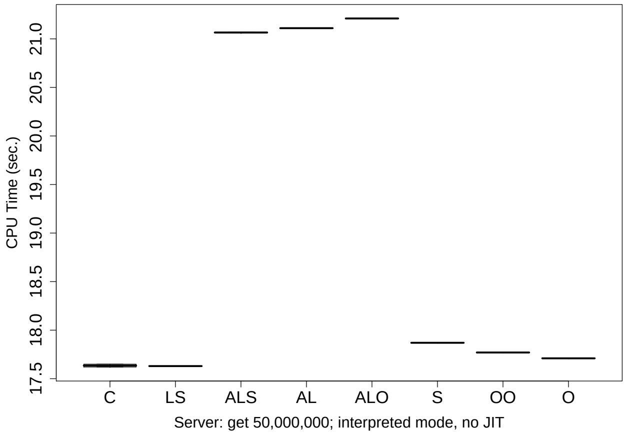


Figure:

Testing `get` method: no inlining mode

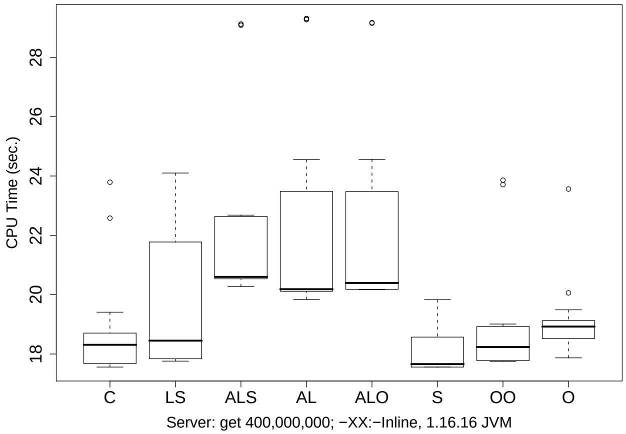


Figure:

Testing `get` method: regular mode

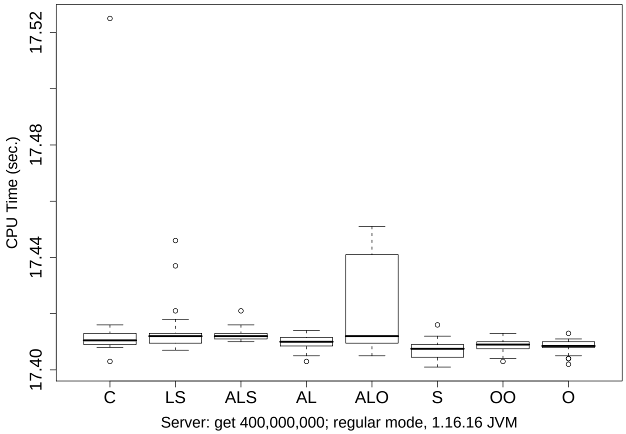


Figure:

Method `getEqual`

- Return boolean.

Example: method `getEqual` on `ArrayList`

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

```
65 aaload
66 if_acmpne 73 (+7)
69 iconst_1
70 goto 74 (+4)
73 iconst_0
74 ireturn
```

Testing `get_equal` method: interpreted mode

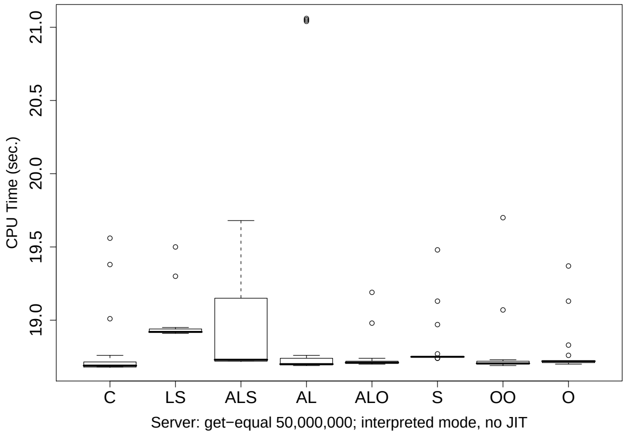


Figure:

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.

Summary of Observations (cont.)

- 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 ClassCast exception thrown?)
  - Explain behavior of other methods (`add`, `set`)
  - Continue trying other JVMs (Jikes RVM, results not shown).

Selected References

1. BRACHA , G. Generics in the java programming language. java.sun.com (2004).
2. LINDHOLM , T., AND YELLIN , F. The Java(TM) Virtual Machine Specification (2nd Edition). Prentice Hall PTR, April 1999.
3. MACHKASOVA , E., ARHELGER , K., AND TRINCIANTE , F. The observer effect of profiling on dynamic java optimizations (poster). In OOPSLA 2009
4. MAYFIELD , E., ROTH , J. K., SELIFONOV, D., DAHLBERG , N., AND MACHKASOVA, E. Optimizing java programs using generic types (poster). In OOPSLA 2007.
5. SUN DEVELOPER NETWORK. The java hotspot performance engine architecture. Sun Microsystem (2007), The java hotspotTM server vm. Sun Microsystem (2008).

```
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
```