Habanero Extreme Scale Software Research Project
Comp215: Java Method Dispatch

Zoran Budimlić (Rice University)
“Always remember that you are absolutely unique. Just like everyone else.”

- Margaret Mead
IList<String> emptyList = List.Empty.create();
IList<String> list = emptyList.add("Alice").add("Bob").add("Charlie");
assertEquals(list.head(), "Charlie");
Java Technology

The Java programming language
The library (JDK)
The Java virtual machine (JVM)
  An instruction set and the meaning of those instructions – the bytecodes
  A binary format – the class file format
  An algorithm to verify the class file
## JVM Data Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reference</td>
<td>Pointer to an object or array</td>
</tr>
<tr>
<td>int</td>
<td>32-bit integer (signed)</td>
</tr>
<tr>
<td>long</td>
<td>64-bit integer (signed)</td>
</tr>
<tr>
<td>float</td>
<td>32-bit floating-point (IEEE 754-1985)</td>
</tr>
<tr>
<td>double</td>
<td>64-bit floating-point (IEEE 754-1985)</td>
</tr>
</tbody>
</table>

**No boolean, char, byte, and short types**

Stack contains only 32-bit and 64-bit data

Conversion instructions
JVM Instruction Set

The Bytecodes
Operations on the operand stack
Variable length
Simple, e.g. iadd
Complex, e.g. new
Symbolic references
201 different instructions
Instruction Types

Arithmetic
Load and store
Type conversion
Object creation and manipulation
Operand stack manipulation
Control transfer
Method invocation and return
Arithmetic Instructions

Operate on the values from the stack
Push the result back onto the stack

Instructions for int, long, float and double

No direct support for byte, short or char types

Handled by int operations and type conversion
**Method Invocation, Return**

**invokevirtual**
Invokes an instance method of an object, dispatching on the (virtual) type of the object.
This is the normal method dispatch in the Java programming language.

**invokeinterface**
Invokes a method that is implemented by an interface.

**invokespecial**
Invokes an instance method requiring special handling.
Instance initialization method, a private method, or a superclass method.

**invokestatic**
Invokes a class (static) method.

**invokedynamic**
allows the language implementor to define custom linkage behavior. Simplifies implementation of non-Java dynamic languages on top of JVM.
Class Information

Instance size
Static (class) variables
Virtual method table
Interface table
Constant pool
Reference to super class
Java Class data structure
## Method Structure

<table>
<thead>
<tr>
<th>Start address</th>
<th>Method length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant pool</td>
<td>Local count</td>
</tr>
</tbody>
</table>

**Information about a method**

- Address
- Length (for the cache)
- Pointer to the constant pool of the class
- Number of arguments and local variables
Object Format

Direct pointer
Handle possible
Return pointer to the class information

Object reference

<table>
<thead>
<tr>
<th>Method vector base</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance variable 1</td>
</tr>
<tr>
<td>Instance variable 2</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Instance variable n</td>
</tr>
</tbody>
</table>
Stack on Method Invocation

SP - stack pointer
VP - variable pointer

```
SP →
arg_2
arg_1
arg_0

Operand stack
...

Context of Caller

Old frame

VP →
var_2
var_1
var_0

Operand stack
...

Context of Caller

var_2
var_1
var_0
```
Your program is a graph

```csharp
IList<String> emptyList = List.Empty.create();
IList<String> list = emptyList.add("Alice").add("Bob").add("Charlie");
assertEquals(list.head(), "Charlie");
```
Method Binding

Determining the method to execute in response to a message

Binding can be accomplished either statically or dynamically

Static Binding:
- Also known as “Early Binding”
- Resolved at compile time
- Resolution based on static type of the object(s)
- Allows compiler optimizations

Dynamic Binding:
- Also known as “Late Binding”
- Resolved at run-time
- Resolution based on the dynamic type of the object(s)
- Uses method dispatch table or Virtual function table
Dynamic dispatch

method call is compiled into `invokevirtual` index

index is always the same for the method

the method structure gets overwritten in the subclass that overrides the method

the JVM looks up the index entry in the virtual method table to find the method information

interfaces have their own tables
Virtual Method Dispatch

```java
public class Foo {
    public void method1(){...};
    public int method2(){...};
    public double method3(){...};
}

public class Bar extends Foo {
    public void method1(){...};
    public double method3(){...};
}

public class Baz extends Foo {
    public void method1(){...};
    public int method2(){...};
}
```
Inline cashing

```csharp
IList<String> emptyList = List.Empty.create();
IList<String> list = emptyList.add("Alice").add("Bob").add("Charlie");
...
assertEquals(list.head(), "Charlie");
```
IList<String> emptyList = List.Empty.create();
IList<String> list = emptyList.add("Alice").add("Bob").add("Charlie");

assertEquals(list.head(), "Charlie");

"Inline cashing" of this method in HotSpot JVM
Deep hierarchies

Virtual method tables can be pretty large
  Memory overhead
Method dispatch can be pretty expensive
Inline caching not always effective
  Truly polymorphic variables
Be careful with your mission-critical code
class List:

    public boolean contains(T element){
        if (head().equals(element)) return true;
        return tail().contains(element);
    }

class List.Empty:

    public boolean contains(T element){
        return false;
    }

if (myList.contains(myElement)) . . .
Tail Recursion

myList.contains() → tail().contains() → Empty().contains()
Tail Recursion

myList.contains()
goTo
tail().contains()
goTo
tail().contains()
goTo
Empty().contains()
Tail recursion and tail call elimination in Java

Not there in Java (yet)

Brian Goetz (Java Language Architect at Oracle):

“... it will eventually get done.”

Other languages have it (Scala, ML, Scheme, ...)

Some of them map onto JVM

Good practice

So when Java finally implements it, or you move to another functional language, you’re ready
Live Coding: Dan’s Log class