



Title: Petter: Programmiersprachen (19.11.2014)
Date: Wed Nov 19 14:20:25 CET 2014
Duration: 96:19 min
Pages: 55

Dispatching Method Calls

Dr. Michael Petter
Winter term 2014

Dispatching - Outline

Dispatching

- ① Motivation
- ② Formal Model
- ③ Quiz
- ④ Dispatching from the Inside

Solutions in Single-Dispatching

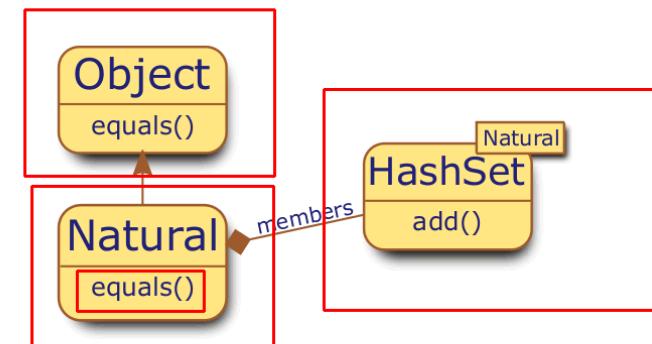
- ① Type introspection
- ② Generic interface

Multi-Dispatching

- ① Formal Model
- ② Multi-Java
- ③ Multi-dispatched compare in Java
- ④ Multi-dispatching in Clojure



Sets of Natural Numbers



Sets of Natural Numbers

```
class Natural {  
    Natural(int n){ number=Math.abs(n); }  
    int number;  
    public boolean equals(Natural n){  
        return n.number == number;  
    }  
}  
...  
Set<Natural> set = new HashSet<>();  
set.add(new Natural(0));  
set.add(new Natural(0));  
System.out.println(set);
```



Sets of Natural Numbers

```
class Natural {  
    Natural(int n){ number=Math.abs(n); }  
    int number;  
    public boolean equals(Natural n){  
        return n.number == number;  
    }  
}  
...  
Set<Natural> set = new HashSet<>();  
set.add(new Natural(0));  
set.add(new Natural(0));  
System.out.println(set);
```



Dispatching Method Calls

The Problem

Example 4 / 31

Generalization

Let's think language independent!



Generalization

Let's think language independent!



`n1.equals(n2);` \implies `equals(n1,n2);`

Dispatching Method Calls

The Problem

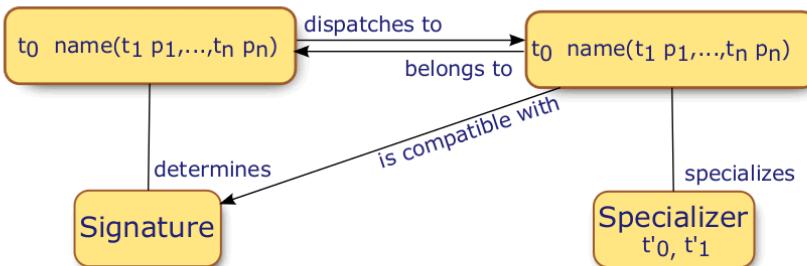
Formal 5 / 31

Dispatching Method Calls

The Problem

Formal 5 / 31

Methods are dynamically dispatched



Dispatching Method Calls

The Problem

Formal

6 / 31

Methods are dynamically dispatched

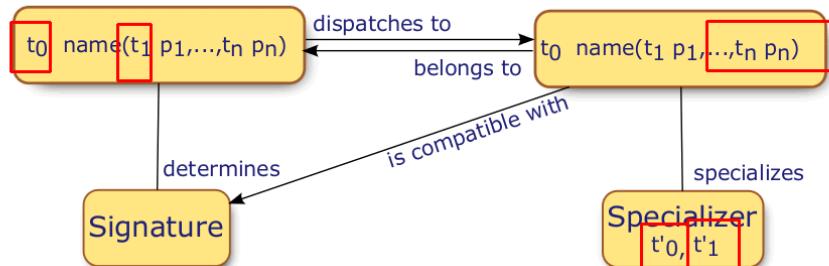


Generic Function

Dynamically dispatched function

Concrete Method

Provides code body for a generic function



Signature

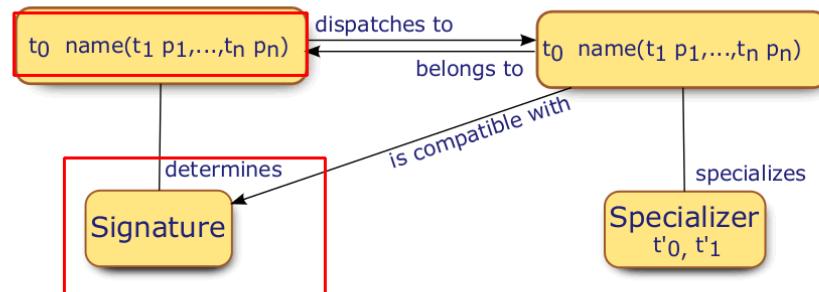
Permissible arguments for calls to generic functions

Methods are dynamically dispatched



Generic Function

Dynamically dispatched function



Dispatching Method Calls

The Problem

Formal

6 / 31

Example: Java [4]



Java determines *generic function signatures* implicitly at each call site from the static types of the arguments.

```
Object o1 = new Natural(1);
Object o2 = new Natural(2);
equals(o1, o2);
```

Signature for call to generic function:

`equals(Object, Object)`

Concrete methods within Natural:

```
boolean equals(Natural n1, Natural n2)
boolean equals(Object o1, Object o2)
```

Dispatching Method Calls

The Problem

Formal

6 / 31

Dispatching Method Calls

The Problem

Formal

7 / 31

Example: Java [4]



Java determines *generic function signatures* implicitly at each call site from the static types of the arguments.

```
Object o1 = new Natural(1);
Object o2 = new Natural(2);
equals(o1,o2);
```

Signature for call to generic function:

```
equals(Object, Object)
```

⚠ Specializer in Java only for return type and first argument

Concrete methods within Natural:

```
boolean equals(Natural n1, Natural n2)
boolean equals(Object o1, Object o2)
boolean equals(Natural o1, Object o2)
```

Mini-Quiz: Java Method Dispatching



```
class A {
    public static void p (Object o) { System.out.println(o); }
    public void m1 (A a) { p("m1(A) in A"); }
    public void m1 () { m1(new B()); }
    private static void m2 (A a) { p("m2(A) in A"); }
    public void m2 () { m2(this); }
}
class B extends A {
    public void m1 (B b) { p("m1(B) in B"); }
    public void m2 (A a) { p("m2(A) in B"); }
    public void m3 () { super.m1(this); }
}
```

```
B b = new B(); A a = b; a.m1(b);
```

Example: Java [4]



Java determines *generic function signatures* implicitly at each call site from the static types of the arguments.

```
Object o1 = new Natural(1);
Object o2 = new Natural(2);
equals(o1,o2);
```

Signature for call to generic function:

```
equals(Object, Object)
```

⚠ Specializer in Java only for return type and first argument

Concrete methods within Natural:

```
boolean equals(Natural n1, Natural n2)
boolean equals(Object o1, Object o2)
boolean equals(Natural o1, Object o2)
```

Mini-Quiz: Java Method Dispatching



```
class A {
    public static void p (Object o) { System.out.println(o); }
    public void m1 (A a) { p("m1(A) in A"); }
    public void m1 () { m1(new B()); }
    private static void m2 (A a) { p("m2(A) in A"); }
    public void m2 () { m2(this); }
}
class B extends A {
    public void m1 (B b) { p("m1(B) in B"); }
    public void m2 (A a) { p("m2(A) in B"); }
    public void m3 () { super.m1(this); }
}
```

```
B b = new B(); A a = b; a.m1(b); m1(A) in A
```

Mini-Quiz: Java Method Dispatching



```
class A {
    public static void p (Object o) { System.out.println(o); }
    public void m1 (A a) { p("m1(A) in A"); }
    public void m1 () { m1(new B()); }
    private static void m2 (A a) { p("m2(A) in A"); }
    public void m2 () { m2(this); }
}
class B extends A {
    public void m1 (B b) { p("m1(B) in B"); }
    public void m2 (A a) { p("m2(A) in B"); }
    public void m3 () { super.m1(this); }
}
```

B b = new B(); A a = b; a.m1(b); m1(A) in A
 B b = new B(); B a = b; b.m1(a);

Dispatching Method Calls

The Problem

Formal 8 / 31

Mini-Quiz: Java Method Dispatching



```
class A {
    public static void p (Object o) { System.out.println(o); }
    public void m1 (A a) { p("m1(A) in A"); }
    public void m1 () { m1(new B()); }
    private static void m2 (A a) { p("m2(A) in A"); }
    public void m2 () { m2(this); }
}
class B extends A {
    public void m1 (B b) { p("m1(B) in B"); }
    public void m2 (A a) { p("m2(A) in B"); }
    public void m3 () { super.m1(this); }
}
```

B b = new B(); A a = b; a.m1(b); m1(A) in A
 B b = new B(); B a = b; b.m1(a); m1(B) in B
 B b = new B(); b.m1(); m1(A) in A

Dispatching Method Calls

The Problem

Formal 8 / 31

Mini-Quiz: Java Method Dispatching



```
class A {
    public static void p (Object o) { System.out.println(o); }
    public void m1 (A a) { p("m1(A) in A"); }
    public void m1 () { m1(new B()); }
    private static void m2 (A a) { p("m2(A) in A"); }
    public void m2 () { m2(this); }
}
class B extends A {
    public void m1 (B b) { p("m1(B) in B"); }
    public void m2 (A a) { p("m2(A) in B"); }
    public void m3 () { super.m1(this); }
}
```

B b = new B(); A a = b; a.m1(b); m1(A) in A
 B b = new B(); B a = b; b.m1(a); m1(B) in B
 B b = new B(); b.m1(); m1(A) in A

Dispatching Method Calls

The Problem

Formal 8 / 31

Mini-Quiz: Java Method Dispatching



```
class A {
    public static void p (Object o) { System.out.println(o); }
    public void m1 (A a) { p("m1(A) in A"); }
    public void m1 () { m1(new B()); }
    private static void m2 (A a) { p("m2(A) in A"); }
    public void m2 () { m2(this); }
}
class B extends A {
    public void m1 (B b) { p("m1(B) in B"); }
    public void m2 (A a) { p("m2(A) in B"); }
    public void m3 () { super.m1(this); }
}
```

B b = new B(); A a = b; a.m1(b); m1(A) in A
 B b = new B(); B a = b; b.m1(a); m1(B) in B
 B b = new B(); b.m1(); m1(A) in A
 B b = new B(); b.m2();

Dispatching Method Calls

The Problem

Formal 8 / 31

Mini-Quiz: Java Method Dispatching



```
class A {  
    public static void p (Object o) { System.out.println(o); }  
    public void m1 (A a) { p("m1(A) in A"); }  
    public void m1 () { m1(new B()); }  
    private static void m2 (A a) { p("m2(A) in A"); }  
    public void m2 () { m2(this); }  
}  
  
class B extends A {  
    public void m1 (B b) { p("m1(B) in B"); }  
    public void m2 (A a) { p("m2(A) in B"); }  
    public void m3 () { super.m1(this); }  
}
```

```
B b = new B(); A a = b; a.m1(b); m1(A) in A  
B b = new B(); B a = b; b.m1(a); m1(B) in B  
B b = new B(); b.m1(); m1(A) in A  
B b = new B(); b.m2(); m2(A) in A
```

Dispatching Method Calls

The Problem

Formal 8 / 31

Mini-Quiz: Java Method Dispatching



```
class A {  
    public static void p (Object o) { System.out.println(o); }  
    public void m1 (A a) { p("m1(A) in A"); }  
    public void m1 () { m1(new B()); }  
    private static void m2 (A a) { p("m2(A) in A"); }  
    public void m2 () { m2(this); }  
}  
  
class B extends A {  
    public void m1 (B b) { p("m1(B) in B"); }  
    public void m2 (A a) { p("m2(A) in B"); }  
    public void m3 () { super.m1(this); }  
}
```

```
B b = new B(); A a = b; a.m1(b); m1(A) in A  
B b = new B(); B a = b; b.m1(a); m1(B) in B  
B b = new B(); b.m1(); m1(A) in A  
B b = new B(); b.m2(); m2(A) in A  
B b = new B(); b.m3(); m1(A) in A
```

Dispatching Method Calls

The Problem

Formal 8 / 31

Mini-Quiz: Java Method Dispatching



```
class A {  
    public static void p (Object o) { System.out.println(o); }  
    public void m1 (A a) { p("m1(A) in A"); }  
    public void m1 () { m1(new B()); }  
    private static void m2 (A a) { p("m2(A) in A"); }  
    public void m2 () { m2(this); }  
}  
  
class B extends A {  
    public void m1 (B b) { p("m1(B) in B"); }  
    public void m2 (A a) { p("m2(A) in B"); }  
    public void m3 () { super.m1(this); }  
}
```

```
B b = new B(); A a = b; a.m1(b); m1(A) in A  
B b = new B(); B a = b; b.m1(a); m1(B) in B  
B b = new B(); b.m1(); m1(A) in A  
B b = new B(); b.m2(); m2(A) in A  
B b = new B(); b.m3();
```

Dispatching Method Calls

The Problem

Formal 8 / 31

So what is happening here?



Let's look at what Java does!

The Java platform as example for state of the art OO systems:

- Static Javac-based compiler
- Dynamic Hotspot JIT-Compiler/Interpreter

Let's watch the following code on its way to the CPU:

```
public static void main(String[] args){  
    Object o1 = new Natural(1);  
    Object o2 = new Natural(2);  
    o1.equals(o2);  
}
```

Dispatching Method Calls

Single-Dispatching

9 / 31

Mini-Quiz: Java Method Dispatching



```
class A {  
    public static void p (Object o) { System.out.println(o); }  
    public void m1 (A a) { p("m1(A) in A"); }  
    public void m1 () { m1(new B()); }  
    private static void m2 (A a) { p("m2(A) in A"); }  
    public void m2 () { m2(this); }  
}  
  
class B extends A {  
    public void m1 (B b) { p("m1(B) in B"); }  
    public void m2 (A a) { p("m2(A) in B"); }  
    public void m3 () { super.m1(this); }  
}  
  
B b = new B(); A a = b; a.m1(b); m1(A) in A  
B b = new B(); B a = b; b.m1(a); m1(B) in B  
B b = new B(); b.m1(); m1(A) in A  
B b = new B(); b.m2(); m2(A) in A  
B b = new B(); b.m3(); m1(A) in A
```

Dispatching Method Calls

The Problem

Formal 8 / 31

Inside the Javac – Prerequisites



Concept of methods being *applicable* for arguments:

```
// true if the given method is applicable to the given arguments  
boolean isApplicable(MemberDefinition m, Type args[]) {  
    // Sanity checks:  
    Type mType = m.getType();  
    if (!mType.isType(TC_METHOD)) return false;  
  
    Type mArgs[] = mType.getArgumentTypes();  
    if (args.length != mArgs.length) return false;  
  
    for (int i = args.length ; --i >= 0 ;)  
        if (!isMoreSpecific(args[i], mArgs[i])) return false;  
    return true;  
}  
boolean isMoreSpecific(Type from, Type to) //... type based specialization
```

Concept of method signatures being *more specific* than others:

```
// true if "best" is in every argument at least as good as other  
boolean isMoreSpecific(MemberDefinition best, MemberDefinition other) {  
    Type bestType = best.getClassDeclaration().getType();  
    Type otherType = other.getClassDeclaration().getType();  
    return isMoreSpecific(bestType, otherType) // type based relation  
        && isApplicable(other, best.getType().getArgumentTypes());  
}
```

Dispatching Method Calls

Single-Dispatching

Inside the Javac 10 / 31

So what is happening here?



Let's look at what Java does!

The Java platform as example for state of the art OO systems:

- Static Javac-based compiler
- Dynamic Hotspot JIT-Compiler/Interpreter

Let's watch the following code on its way to the CPU:

```
public static void main(String[] args){  
    Object o1 = new Natural(1);  
    Object o2 = new Natural(2);  
    o1.equals(o2);  
}
```

Dispatching Method Calls

Single-Dispatching

9 / 31

Inside the Javac



```
MemberDefinition matchMethod(Environment env, ClassDefinition accessor,  
    Identifier methodName, Type[] argumentTypes) throws ... {  
    // A tentative maximally specific method.  
    MemberDefinition tentative = null;  
    // A list of other methods which may be maximally specific too.  
    List candidateList = null;  
    // Get all the methods inherited by this class which have the name 'methodName'  
    for (MemberDefinition method : allMethods.lookupName(methodName)) {  
        // See if this method is applicable.  
        if (!env.isApplicable(method, argumentTypes)) continue;  
        // See if this method is accessible.  
        if ((accessor != null) && (!accessor.canAccess(env, method))) continue;  
        if ((method == null) || (env.isMoreSpecific(method, tentative)))  
            // 'method' becomes our tentative maximally specific match.  
            tentative = method;  
        else { // If this method could possibly be another maximally specific  
            // method, add it to our list of other candidates.  
            if (!env.isMoreSpecific(tentative, method)) {  
                if (candidateList == null) candidateList = new ArrayList();  
                candidateList.add(method);  
            } } }  
    if (tentative != null && candidateList != null)  
        // Find out if our 'tentative' match is a uniquely maximally specific.  
        for (MemberDefinition method : candidateList)  
            if (!env.isMoreSpecific(tentative, method))  
                throw new AmbiguousMember(tentative, method);  
    return tentative;  
}
```

Dispatching Method Calls

Single-Dispatching

Inside the Javac 11 / 31

Bytecode



- ~> matchMethod returns the statically most specific signature
- ~> Codegeneration hardcodes invokevirtual with this signature

```
Code:  
0: new #4; //class Natural  
3: dup  
4: iconst_1  
5: invokespecial #5; //Method "<init>":(I)V  
8: astore_1  
9: new #4; //class Natural  
12: dup  
13: iconst_2  
14: invokespecial #5; //Method "<init>":(I)V  
17: astore_2  
18: aload_1  
19: aload_2  
20: invokevirtual #6; //Method java/lang/Object.equals:(Ljava/lang/Object;)Z  
23: pop  
24: return
```

? What is the semantics of invokevirtual?

Dispatching Method Calls

Single-Dispatching

Inside the Javac 12 / 31

Inside the Hotspot



```
void LinkResolver::resolve_method(methodHandle& resolved_method, KlassHandle resolved_klass,  
                                  Symbol* method_name, Symbol* method_signature,  
                                  KlassHandle current_klass) {  
  
    // 1. check if klass is not interface  
    if (resolved_klass->is_interface()) ;//... throw "Found interface, but class was expected"  
  
    // 2. lookup method in resolved klass and its super klasses  
    lookup_method_in_klasses(resolved_method, resolved_klass, method_name, method_signature);  
    // calls klass::lookup_method() -> next slide  
  
    if (resolved_method.is_null()) { // not found in the class hierarchy  
        // 3. lookup method in all the interfaces implemented by the resolved klass  
        lookup_method_in_interfaces(resolved_method, resolved_klass, method_name, method_signature);  
  
        if (resolved_method.is_null()) {  
            // JSR 292: see if this is an implicitly generated method MethodHandle.invoke(*...)  
            lookup_implicit_method(resolved_method, resolved_klass, method_name, method_signature, ...);  
        }  
  
        if (resolved_method.is_null()) { // 4. method lookup failed  
            // ... throw java_lang_NoSuchMethodError()  
        }  
  
        // 5. check if method is concrete  
        if (resolved_method->is_abstract() && !resolved_klass->is_abstract()) {  
            // ... throw java_lang_AbstractMethodError()  
        }  
  
        // 6. access checks, etc.
```

Dispatching Method Calls

Single-Dispatching

Inside the Hotspot 13 / 31

Bytecode



- ~> matchMethod returns the statically most specific signature
- ~> Codegeneration hardcodes invokevirtual with this signature

```
Code:  
0: new #4; //class Natural  
3: dup  
4: iconst_1  
5: invokespecial #5; //Method "<init>":(I)V  
8: astore_1  
9: new #4; //class Natural  
12: dup  
13: iconst_2  
14: invokespecial #5; //Method "<init>":(I)V  
17: astore_2  
18: aload_1  
19: aload_2  
20: invokevirtual #6; //Method java/lang/Object.equals:(Ljava/lang/Object;)Z  
23: pop  
24: return
```

? What is the semantics of invokevirtual?

~> Check the runtime interpreter: Hotspot VM calls `resolve_method!`

Dispatching Method Calls

Single-Dispatching

Inside the Javac 12 / 31

Inside the Hotspot VM



The method lookup recursively traverses the super class chain:

```
MethodDesc* klass::lookup_method(Symbol* name, Symbol* signature) {  
    for (KlassDesc* klas = as_klassOop(); klas != NULL; klas = klas::cast(klas)->super()) {  
        MethodDesc* method = klass::cast(klass)->find_method(name, signature);  
        if (method != NULL) return method;  
    }  
    return NULL;  
}
```

Dispatching Method Calls

Single-Dispatching

Inside the Hotspot 14 / 31

Inside the Hotspot

```
void LinkResolver::resolve_method(methodHandle& resolved_method, KlassHandle resolved_klass,
    Symbol* method_name, Symbol* method_signature,
    KlassHandle current_klass) {

    // 1. check if klass is not interface
    if (resolved_klass->is_interface()) ;//... throw "Found interface, but class was expected"

    // 2. lookup method in resolved klass and its super klasses
    lookup_method_in_klasses(resolved_method, resolved_klass, method_name, method_signature);
    // calls klass::lookup_method() -> next slide

    if (resolved_method.is_null()) { // not found in the class hierarchy
        // 3. lookup method in all the interfaces implemented by the resolved klass
        lookup_method_in_interfaces(resolved_method, resolved_klass, method_name, method_signature);

        if (resolved_method.is_null()) {
            // JSR 292: see if this is an implicitly generated method MethodHandle.invoke(*...)
            lookup_implicit_method(resolved_method, resolved_klass, method_name, method_signature, cu);
        }

        if (resolved_method.is_null()) { // 4. method lookup failed
            // ... throw java_lang_NoSuchMethodError()
        }
    }

    // 5. check if method is concrete
    if (resolved_method->is_abstract() && !resolved_klass->is_abstract()) {
        // ... throw java_lang_AbstractMethodError()
    }

    // 6. access checks, etc.
}
```

Dispatching Method Calls

Single-Dispatching

Inside the Hotspot

13 / 31



Inside the Hotspot

```
MethodDesc* klass::find_method(ObjArrayDesc* methods, Symbol* name, Symbol* signature) {
    int len = methods->length();
    // methods are sorted, so do binary search
    int i, l = 0, h = len - 1;
    while (l <= h) {
        int mid = (l + h) >> 1;
        MethodDesc* m = (MethodDesc*)methods->obj_at(mid);
        int res = m->name()->fast_compare(name);
        if (res == 0) {
            // found matching name; do linear search to find matching signature
            // first, quick check for common case
            if (m->signature() == signature) return m;
            // search downwards through overloaded methods
            for (i = mid - 1; i >= l; i--) {
                MethodDesc* m = (MethodDesc*)methods->obj_at(i);
                if (m->name() != name) break;
                if (m->signature() == signature) return m;
            }
            // search upwards
            for (i = mid + 1; i <= h; i++) {
                MethodDesc* m = (MethodDesc*)methods->obj_at(i);
                if (m->name() != name) break;
                if (m->signature() == signature) return m;
            }
        }
        return NULL; // not found
    } else if (res < 0) l = mid + 1;
    else h = mid - 1;
}
return NULL;
}
```

Dispatching Method Calls

Single-Dispatching

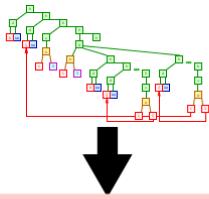
Inside the Hotspot

15 / 31



Summary

Compile Time



JavaC

Matches a method call expression *statically* to the *most specific* method signature via `matchMethod(...)`

Runtime

```
Code:
1: 0x0000000000000000 // class Natural
2: 0x0000000000000001
3: 0x0000000000000002
4: 0x0000000000000003
5: 0x0000000000000004 // Method <init>()V
6: 0x0000000000000005
7: 0x0000000000000006
8: 0x0000000000000007
9: 0x0000000000000008 // Class Natural
10: 0x0000000000000009
11: 0x000000000000000A
12: 0x000000000000000B
13: 0x000000000000000C // Method <init>()V
14: 0x000000000000000D
15: 0x000000000000000E
16: 0x000000000000000F
17: 0x000000000000000G
18: 0x000000000000000H
19: 0x000000000000000I
20: 0x000000000000000J // Method hashCode()I
21: 0x000000000000000K
22: 0x000000000000000L
23: 0x000000000000000M
24: 0x000000000000000N
25: 0x000000000000000O
26: 0x000000000000000P
27: 0x000000000000000Q
28: 0x000000000000000R
29: 0x000000000000000S
30: 0x000000000000000T
31: 0x000000000000000U
32: 0x000000000000000V
33: 0x000000000000000W
34: 0x000000000000000X
35: 0x000000000000000Y
36: 0x000000000000000Z
37: 0x000000000000000a
38: 0x000000000000000b
39: 0x000000000000000c
40: 0x000000000000000d
41: 0x000000000000000e
42: 0x000000000000000f
43: 0x000000000000000g
44: 0x000000000000000h
45: 0x000000000000000i
46: 0x000000000000000j
47: 0x000000000000000k
48: 0x000000000000000l
49: 0x000000000000000m
50: 0x000000000000000n
51: 0x000000000000000o
52: 0x000000000000000p
53: 0x000000000000000q
54: 0x000000000000000r
55: 0x000000000000000s
56: 0x000000000000000t
57: 0x000000000000000u
58: 0x000000000000000v
59: 0x000000000000000w
60: 0x000000000000000x
61: 0x000000000000000y
62: 0x000000000000000z
63: 0x000000000000000A
64: 0x000000000000000B
65: 0x000000000000000C
66: 0x000000000000000D
67: 0x000000000000000E
68: 0x000000000000000F
69: 0x000000000000000G
70: 0x000000000000000H
71: 0x000000000000000I
72: 0x000000000000000J
73: 0x000000000000000K
74: 0x000000000000000L
75: 0x000000000000000M
76: 0x000000000000000N
77: 0x000000000000000O
78: 0x000000000000000P
79: 0x000000000000000Q
80: 0x000000000000000R
81: 0x000000000000000S
82: 0x000000000000000T
83: 0x000000000000000U
84: 0x000000000000000V
85: 0x000000000000000W
86: 0x000000000000000X
87: 0x000000000000000Y
88: 0x000000000000000Z
89: 0x000000000000000a
90: 0x000000000000000b
91: 0x000000000000000c
92: 0x000000000000000d
93: 0x000000000000000e
94: 0x000000000000000f
95: 0x000000000000000g
96: 0x000000000000000h
97: 0x000000000000000i
98: 0x000000000000000j
99: 0x000000000000000k
100: 0x000000000000000l
101: 0x000000000000000m
102: 0x000000000000000n
103: 0x000000000000000o
104: 0x000000000000000p
105: 0x000000000000000q
106: 0x000000000000000r
107: 0x000000000000000s
108: 0x000000000000000t
109: 0x000000000000000u
110: 0x000000000000000v
111: 0x000000000000000w
112: 0x000000000000000x
113: 0x000000000000000y
114: 0x000000000000000z
115: 0x000000000000000A
116: 0x000000000000000B
117: 0x000000000000000C
118: 0x000000000000000D
119: 0x000000000000000E
120: 0x000000000000000F
121: 0x000000000000000G
122: 0x000000000000000H
123: 0x000000000000000I
124: 0x000000000000000J
125: 0x000000000000000K
126: 0x000000000000000L
127: 0x000000000000000M
128: 0x000000000000000N
129: 0x000000000000000O
130: 0x000000000000000P
131: 0x000000000000000Q
132: 0x000000000000000R
133: 0x000000000000000S
134: 0x000000000000000T
135: 0x000000000000000U
136: 0x000000000000000V
137: 0x000000000000000W
138: 0x000000000000000X
139: 0x000000000000000Y
140: 0x000000000000000Z
141: 0x000000000000000a
142: 0x000000000000000b
143: 0x000000000000000c
144: 0x000000000000000d
145: 0x000000000000000e
146: 0x000000000000000f
147: 0x000000000000000g
148: 0x000000000000000h
149: 0x000000000000000i
150: 0x000000000000000j
151: 0x000000000000000k
152: 0x000000000000000l
153: 0x000000000000000m
154: 0x000000000000000n
155: 0x000000000000000o
156: 0x000000000000000p
157: 0x000000000000000q
158: 0x000000000000000r
159: 0x000000000000000s
160: 0x000000000000000t
161: 0x000000000000000u
162: 0x000000000000000v
163: 0x000000000000000w
164: 0x000000000000000x
165: 0x000000000000000y
166: 0x000000000000000z
167: 0x000000000000000A
168: 0x000000000000000B
169: 0x000000000000000C
170: 0x000000000000000D
171: 0x000000000000000E
172: 0x000000000000000F
173: 0x000000000000000G
174: 0x000000000000000H
175: 0x000000000000000I
176: 0x000000000000000J
177: 0x000000000000000K
178: 0x000000000000000L
179: 0x000000000000000M
180: 0x000000000000000N
181: 0x000000000000000O
182: 0x000000000000000P
183: 0x000000000000000Q
184: 0x000000000000000R
185: 0x000000000000000S
186: 0x000000000000000T
187: 0x000000000000000U
188: 0x000000000000000V
189: 0x000000000000000W
190: 0x000000000000000X
191: 0x000000000000000Y
192: 0x000000000000000Z
193: 0x000000000000000a
194: 0x000000000000000b
195: 0x000000000000000c
196: 0x000000000000000d
197: 0x000000000000000e
198: 0x000000000000000f
199: 0x000000000000000g
200: 0x000000000000000h
201: 0x000000000000000i
202: 0x000000000000000j
203: 0x000000000000000k
204: 0x000000000000000l
205: 0x000000000000000m
206: 0x000000000000000n
207: 0x000000000000000o
208: 0x000000000000000p
209: 0x000000000000000q
210: 0x000000000000000r
211: 0x000000000000000s
212: 0x000000000000000t
213: 0x000000000000000u
214: 0x000000000000000v
215: 0x000000000000000w
216: 0x000000000000000x
217: 0x000000000000000y
218: 0x000000000000000z
219: 0x000000000000000A
220: 0x000000000000000B
221: 0x000000000000000C
222: 0x000000000000000D
223: 0x000000000000000E
224: 0x000000000000000F
225: 0x000000000000000G
226: 0x000000000000000H
227: 0x000000000000000I
228: 0x000000000000000J
229: 0x000000000000000K
230: 0x000000000000000L
231: 0x000000000000000M
232: 0x000000000000000N
233: 0x000000000000000O
234: 0x000000000000000P
235: 0x000000000000000Q
236: 0x000000000000000R
237: 0x000000000000000S
238: 0x000000000000000T
239: 0x000000000000000U
240: 0x000000000000000V
241: 0x000000000000000W
242: 0x000000000000000X
243: 0x000000000000000Y
244: 0x000000000000000Z
245: 0x000000000000000a
246: 0x000000000000000b
247: 0x000000000000000c
248: 0x000000000000000d
249: 0x000000000000000e
250: 0x000000000000000f
251: 0x000000000000000g
252: 0x000000000000000h
253: 0x000000000000000i
254: 0x000000000000000j
255: 0x000000000000000k
256: 0x000000000000000l
257: 0x000000000000000m
258: 0x000000000000000n
259: 0x000000000000000o
260: 0x000000000000000p
261: 0x000000000000000q
262: 0x000000000000000r
263: 0x000000000000000s
264: 0x000000000000000t
265: 0x000000000000000u
266: 0x000000000000000v
267: 0x000000000000000w
268: 0x000000000000000x
269: 0x000000000000000y
270: 0x000000000000000z
271: 0x000000000000000A
272: 0x000000000000000B
273: 0x000000000000000C
274: 0x000000000000000D
275: 0x000000000000000E
276: 0x000000000000000F
277: 0x000000000000000G
278: 0x000000000000000H
279: 0x000000000000000I
280: 0x000000000000000J
281: 0x000000000000000K
282: 0x000000000000000L
283: 0x000000000000000M
284: 0x000000000000000N
285: 0x000000000000000O
286: 0x000000000000000P
287: 0x000000000000000Q
288: 0x000000000000000R
289: 0x000000000000000S
290: 0x000000000000000T
291: 0x000000000000000U
292: 0x000000000000000V
293: 0x000000000000000W
294: 0x000000000000000X
295: 0x000000000000000Y
296: 0x000000000000000Z
297: 0x000000000000000a
298: 0x000000000000000b
299: 0x000000000000000c
300: 0x000000000000000d
301: 0x000000000000000e
302: 0x000000000000000f
303: 0x000000000000000g
304: 0x000000000000000h
305: 0x000000000000000i
306: 0x000000000000000j
307: 0x000000000000000k
308: 0x000000000000000l
309: 0x000000000000000m
310: 0x000000000000000n
311: 0x000000000000000o
312: 0x000000000000000p
313: 0x000000000000000q
314: 0x000000000000000r
315: 0x000000000000000s
316: 0x000000000000000t
317: 0x000000000000000u
318: 0x000000000000000v
319: 0x000000000000000w
320: 0x000000000000000x
321: 0x000000000000000y
322: 0x000000000000000z
323: 0x000000000000000A
324: 0x000000000000000B
325: 0x000000000000000C
326: 0x000000000000000D
327: 0x000000000000000E
328: 0x000000000000000F
329: 0x000000000000000G
330: 0x000000000000000H
331: 0x000000000000000I
332: 0x000000000000000J
333: 0x000000000000000K
334: 0x000000000000000L
335: 0x000000000000000M
336: 0x000000000000000N
337: 0x000000000000000O
338: 0x000000000000000P
339: 0x000000000000000Q
340: 0x000000000000000R
341: 0x000000000000000S
342: 0x000000000000000T
343: 0x000000000000000U
344: 0x000000000000000V
345: 0x000000000000000W
346: 0x000000000000000X
347: 0x000000000000000Y
348: 0x000000000000000Z
349: 0x000000000000000a
350: 0x000000000000000b
351: 0x000000000000000c
352: 0x000000000000000d
353: 0x000000000000000e
354: 0x000000000000000f
355: 0x000000000000000g
356: 0x000000000000000h
357: 0x000000000000000i
358: 0x000000000000000j
359: 0x000000000000000k
360: 0x000000000000000l
361: 0x000000000000000m
362: 0x000000000000000n
363: 0x000000000000000o
364: 0x000000000000000p
365: 0x000000000000000q
366: 0x000000000000000r
367: 0x000000000000000s
368: 0x000000000000000t
369: 0x000000000000000u
370: 0x000000000000000v
371: 0x000000000000000w
372: 0x000000000000000x
373: 0x000000000000000y
374: 0x000000000000000z
375: 0x000000000000000A
376: 0x000000000000000B
377: 0x000000000000000C
378: 0x000000000000000D
379: 0x000000000000000E
380: 0x000000000000000F
381: 0x000000000000000G
382: 0x000000000000000H
383: 0x000000000000000I
384: 0x000000000000000J
385: 0x000000000000000K
386: 0x000000000000000L
387: 0x000000000000000M
388: 0x000000000000000N
389: 0x000000000000000O
390: 0x000000000000000P
391: 0x000000000000000Q
392: 0x000000000000000R
393: 0x000000000000000S
394: 0x000000000000000T
395: 0x000000000000000U
396: 0x000000000000000V
397: 0x000000000000000W
398: 0x000000000000000X
399: 0x000000000000000Y
400: 0x000000000000000Z
401: 0x000000000000000a
402: 0x000000000000000b
403: 0x000000000000000c
404: 0x000000000000000d
405: 0x000000000000000e
406: 0x000000000000000f
407: 0x000000000000000g
408: 0x000000000000000h
409: 0x00000000
```

Introspection

```
class Natural {  
    Natural(int n){ number=Math.abs(n); }  
    int number;  
    public boolean equals(Object n){  
        if (!(n instanceof Natural)) return false;  
        return ((Natural)n).number == number;  
    }  
}  
  
...  
Set<Natural> set = new HashSet<>();  
set.add(new Natural(0));  
set.add(new Natural(0));  
System.out.println(set);
```



Introspection

```
class Natural {  
    Natural(int n){ number=Math.abs(n); }  
    int number;  
    public boolean equals(Object n){  
        if (!(n instanceof Natural)) return false;  
        return ((Natural)n).number == number;  
    }  
}  
  
...  
Set<Natural> set = new HashSet<>();  
set.add(new Natural(0));  
set.add(new Natural(0));  
System.out.println(set);
```



```
>$ java Natural  
[0]
```

✓ Works ⚠ but bothers programmer with type safety

Introspection

```
class Natural {  
    Natural(int n){ number=Math.abs(n); }  
    int number;  
    public boolean equals(Object n){  
        if (!(n instanceof Natural)) return false;  
        return ((Natural)n).number == number;  
    }  
}  
  
...  
Set<Natural> set = new HashSet<>();  
set.add(new Natural(0));  
set.add(new Natural(0));  
System.out.println(set);
```

```
>$ java Natural  
[0]
```

✓ Works ⚠ but bothers programmer with type safety
⚠ and is only available for languages with type introspection



Generic Programming

```
interface Equalizable<T>{  
    boolean equals(T other);  
}  
class Natural implements Equalizable<Natural> {  
    Natural(int n){ number=Math.abs(n); }  
    int number;  
    public boolean equals(Natural n){  
        return n.number == number;  
    }  
}  
  
...  
EqualizableAwareSet<Natural> set = new MyHashSet<>();  
set.add(new Natural(0));  
set.add(new Natural(0));  
System.out.println(set);
```



Generic Programming



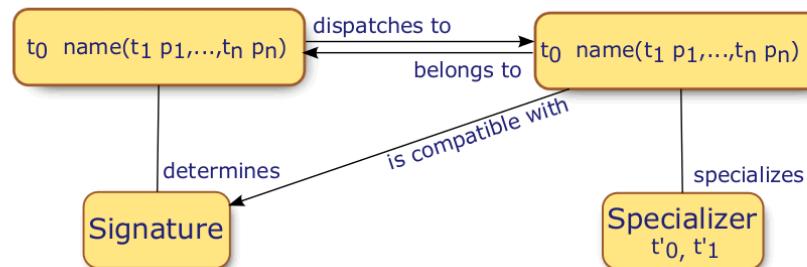
```
interface Equalizable<T>{
    boolean equals(T other);
}

class Natural implements Equalizable<Natural> {
    Natural(int n){ number=Math.abs(n); }
    int number;
    public boolean equals(Natural n){
        return n.number == number;
    }
}

...
EqualizableAwareSet<Natural> set = new MyHashSet<>();
set.add(new Natural(0));
set.add(new Natural(0));
System.out.println(set);
```

⚠ but needs another Set implementation and...

Formal Model of Multi-Dispatching [7]



Generic Programming



```
interface Equalizable<T>{
    boolean equals(T other);
}

class Natural implements Equalizable<Natural> {
    Natural(int n){ number=Math.abs(n); }
    int number;
    public boolean equals(Natural n){
        return n.number == number;
    }
}

...
EqualizableAwareSet<Natural> set = new MyHashSet<>();
set.add(new Natural(0));
set.add(new Natural(0));
System.out.println(set);
```

⚠ but needs another Set implementation and... ⚠ does not compile

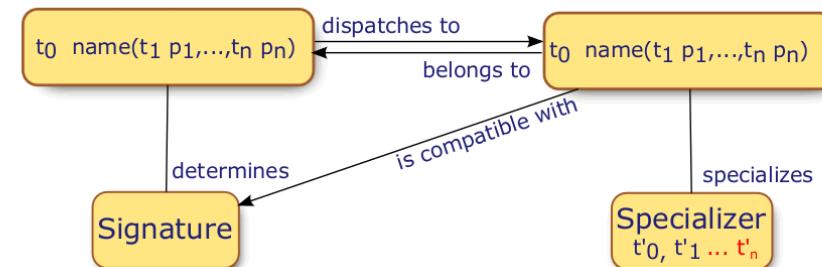
```
>$ javac Natural.java
Natural.java:2: error: name clash: equals(T) in Equalizable and equals(Object)
in Object have the same erasure, yet neither overrides the other
```

Formal Model of Multi-Dispatching [7]



Idea

Introduce Specializers for all parameters

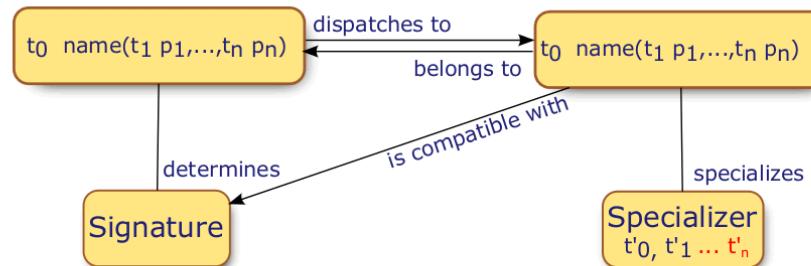


Formal Model of Multi-Dispatching [7]



Idea

Introduce Specializers for all parameters



How it works

- ① Specializers as subtype annotations to parameter types
- ② Dispatcher selects *Most Specific* Concrete Method

Natural Numbers in Multi-Java [3]



```
class Natural {  
    public Natural(int n){ number=Math.abs(n); }  
    private int number;  
    public boolean equals(Object@Natural n){  
        return n.number == number;  
    }  
    ...  
    Set<Natural> set = new HashSet<>();  
    set.add(new Natural(0));  
    set.add(new Natural(0));  
    System.out.println(set);  
}
```



Implications of the implementation

Type-Checking

- ➊ Typechecking families of concrete methods introduces checking the existence of unique most specific methods for all *valid visible type tuples*.
- ➋ Multiple-Inheritance or interfaces as specializers introduce ambiguities, and thus induce runtime ambiguity exceptions

Code-Generation

- ➌ Specialized methods generated separately
- ➍ Dispatcher method calls specialized methods
- ➎ Order of the dispatch tests ensures to find the most specialized method

Performance penalty

The runtime-penalty for multi-dispatching is number of parameters of a multi-method many `instanceof` tests.

Natural Numbers Behind the Scenes

```
>$ javap -c Natural
```

```
public boolean equals(java.lang.Object);  
Code:  
 0:  aload_1  
 1:  instanceof      #2; //class Natural  
 4:  ifeq   16  
 7:  aload_0  
 8:  aload_1  
 9:  checkcast      #2; //class Natural  
12:  invokespecial  #28; //Method equals$body3$0:(LNatural;)Z  
15:  ireturn  
16:  aload_0  
17:  aload_1  
18:  invokespecial  #31; //Method equals$body3$1:(LObject;)Z  
21:  ireturn
```

Clojure

... is a lisp dialect for the JVM with:

- Prefix notation
- () – Brackets for lists
- :: – Userdefined keyword constructor ::keyword
- [] – Vector constructor
- fn – Creates a lambda expression
(fn [x y] (+ x y))
- derive – Generates hierarchical relationships
(derive ::child ::parent)
- defmulti – Creates new generic method
(defmulti name dispatch-fn)
- defmethod – Creates new concrete method
(defmethod name dispatch-val &fn-tail)

How about a natively dynamically dispatched language?

Principle of Multidispatching in Clojure



```
(derive ::child ::parent)

(defmulti fun (fn [a b] [a b]))
(defmethod fun [::child ::child] [a b] ("child equals"))
(defmethod fun [::parent ::parent] [a b] ("parent equals"))

(pr (fun ::child ::child))
```

Natural Numbers in Clojure



Instanciating the dispatching function as follows approximates the desired behaviour in the Java Natural Numbers case:

```
(defmulti equ (fn [a b] [(:Class a) (:Class b)]))
(defmethod equ [::Natural ::Natural] [c1 c2]
  (= (:number c1) (:number c2)))
(defmethod equ [::default] [x y] false)
(defn natural [i] {::Class ::Natural ::number i})
(defn object [] {::Class ::Object})
(def n1 (natural 42))
(def n2 (natural 42))
(def o (object ))
(pr (equ n1 n2))
(pr (equ o n1))
```

More Creative dispatching in Clojure



```
(defn salary [amount]
  (cond (< amount 600)  ::poor
        (>= amount 5000) ::boss
        :else           ::assi))

(defrecord UniPerson [name wage])

(defmulti print (fn [person] (salary (:wage person)) ))
(defmethod print ::poor [person] (str "HiWi " (:name person)))
(defmethod print ::assi [person] (str "Dr. " (:name person)))
(defmethod print ::boss [person] (str "Prof. " (:name person)))

(pr (print (UniPerson. "Simon" 4000)))
(pr (print (UniPerson. "Stefan" 500)))
(pr (print (UniPerson. "Seidl" 6000)))
```

Dispatching Method Calls

Multi-Dispatching

Multi-dispatched compare

28 / 31

Multidispatching



Pro

- Generalization of an established technique
- Directly solves problem
- Eliminates boilerplate code
- Compatible with modular compilation/type checking

Con

- Counters privileged 1st parameter
- Runtime overhead
- New exceptions when used with multi-inheritance
- Most Specific Method* ambiguous

Other Solutions (extract)

- Dylan
- Scala

Lessons Learned



Lessons Learned

- Dynamically dispatched methods are complex interaction of static and dynamic techniques
- Single Dispatching as in major OO-Languages
- Making use of Open Source Compilers
- Multi Dispatching generalizes single dispatching
- Multi Dispatching Java
- Multi Dispatching Clojure

Dispatching Method Calls

Multi-Dispatching

Multi-dispatched compare

30 / 31

Further reading...



- [1] hotspot/src/share/vm/interpreter/linkResolver.cpp.
OpenJDK 7 Hotspot JIT VM.
<http://hg.openjdk.java.net/jdk7/jdk7>.
- [2] jdk/src/share/classes/sun/tools/java/ClassDefinition.java.
OpenJDK 7 Javac.
<http://hg.openjdk.java.net/jdk7/jdk7>.
- [3] C. Clifton, T. Millstein, G. T. Leavens, and C. Chambers.
MultiJava: Design rationale, compiler implementation, and applications.
ACM Transactions on Programming Languages and Systems (TOPLAS), May 2006.
- [4] J. Gosling, B. Joy, G. Steele, and G. Bracha.
The Java Language Specification, Third Edition.
Addison-Wesley Longman, Amsterdam, 3 edition, June 2005.
- [5] S. Halloway.
Programming Clojure.
Pragmatic Bookshelf, 1st edition, 2009.
- [6] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley.
The Java Virtual Machine Specification.
Addison-Wesley Professional, Java SE7 edition, 2013.
- [7] R. Muschevici, A. Potanin, E. Tempero, and J. Noble.
Multiple dispatch in practice.
23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications (OOPSLA), September 2008.

Dispatching Method Calls

Further materials

31 / 31