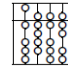
Script generated by TTT

Title: Simon: Programmiersprachen (30.11.2012)

Date: Fri Nov 30 11:03:50 CET 2012

Duration: 83:52 min

Pages: 47



TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Programming Languages

Dispatching Method Calls

Dr. Axel Simon and Dr. Michael Petter
Winter term 2012

Dispatching - Outline

Dispatching

1. Example
2. Formal Model
3. Quiz

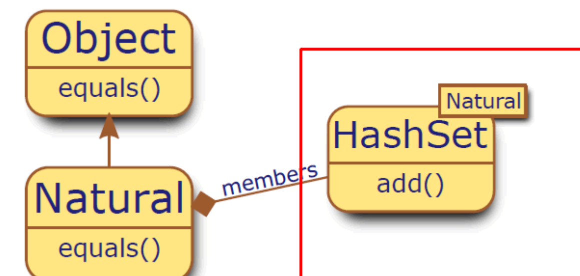
Solutions in Single-Dispatching

1. Type introspection
2. Generic interface
3. Visitor-Pattern

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);
```

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

⚠ Why? Is HashSet buggy?

Generalization

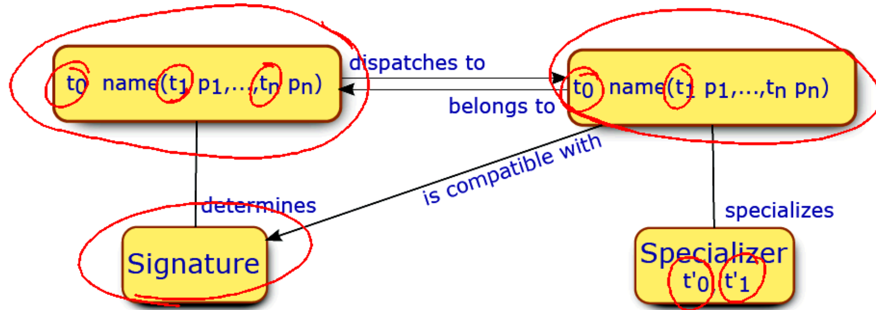
Let's think language independent!

Generalization

Let's think language independent!

`n1.equals(n2);` $\Longrightarrow$ `equals(n1,n2);`

Methods are *dynamically dispatched*



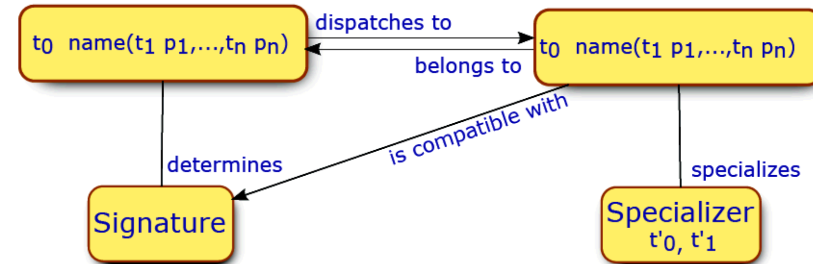
Methods are *dynamically dispatched*

Generic Function

Dynamically dispatched function

Concrete Method

Provides code body for a generic function



Signature

Permissible arguments for calls to generic functions

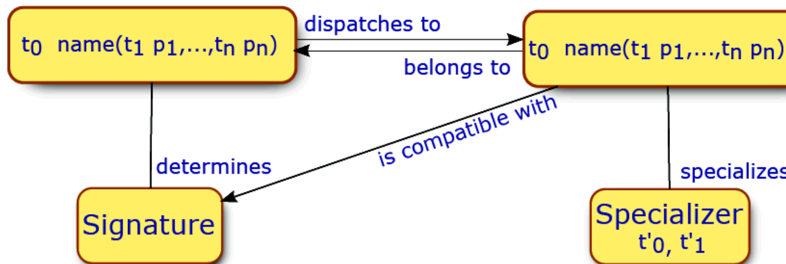
Methods are *dynamically dispatched*

Generic Function

Dynamically dispatched function

Concrete Method

Provides code body for a generic function



Signature

Permissible arguments for calls to generic functions

Specializer

Specialized types to be matched at the call

Example: Java [2]

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)
```

Example: Java [2]

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);
```

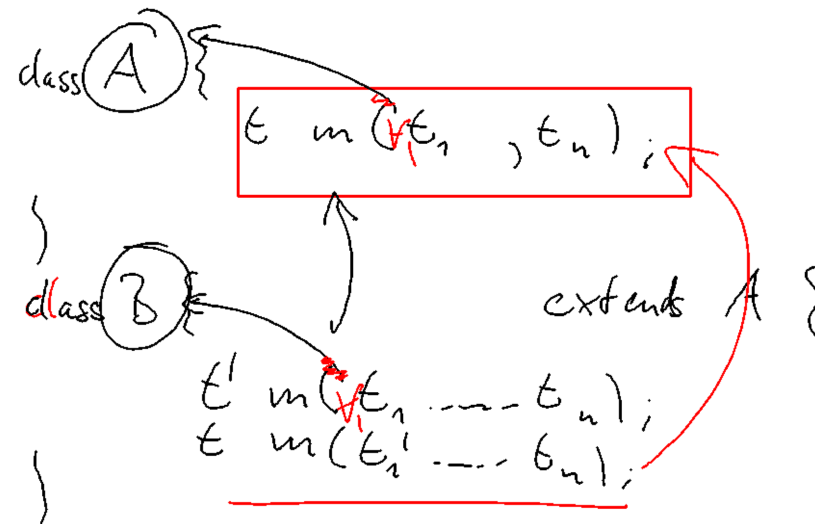
Signature for call to generic function:

```
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)
```

Example: Java [2]

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)
```

Dispatching Method Calls The Problem Formal 7 / 22

Example: Java [2]

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

⚠ and static methods are not specialized at all

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

```java
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);         }
}
```

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


Mini-Quiz: Java Method Dispatching

```java
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);         }
}
```

```java
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
```


Mini-Quiz: Java Method Dispatching

```java
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 () {  this   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);         }
}
```

```java
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();
```


Mini-Quiz: Java Method Dispatching

```java
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);         }
}
```

```java
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
```

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
```


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();
```


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
```


So what to do with Single-Dispatching?

Mainstream languages support specialization of first parameter:

C++, Java, C#, Smalltalk, Lisp

So how do we solve the `equals()` problem?

1. introspection
2. generic programming
3. uses of single-dispatching patterns

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

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);
```

n.getClass(). ==

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

✓ Works ⚠ but burdens 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 burdens 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

```
class Residue extends Natural implements Equalizable<Residue>{
  Residue(int n,int m){ number=Math.abs(n); mod=m; }
  int number;
  int mod;
  public boolean equals(Natural n){
    return n.number == number%mod;
  }
  public boolean equals(Residue r){
    return r.number%mod == number%mod;
  }
}
...
  EqualizableAwareSet<Natural> set = new MyHashSet<>();
  set.add(new Natural(4));
  set.add(new Residue(4,4));
  System.out.println(set);
```


Generic Programming

```
class Residue extends Natural implements Equalizable<Residue>{
  Residue(int n,int m){ number=Math.abs(n); mod=m; }
  int number;
  int mod;
  public boolean equals(Natural n){
    return n.number == number%mod;
  }
  public boolean equals(Residue r){
    return r.number%mod == number%mod;
  }
}
...
  EqualizableAwareSet<Natural> set = new MyHashSet<>();
  set.add(new Natural(4));
  set.add(new Residue(4,4));
  System.out.println(set);
```

```
>$ java Residue
[4]
```

⚠ Result now depends on the exact syntax of the `equals` call

Visitor-Pattern

```
abstract class NaturalVisitor<T> {
  abstract T visit(Natural n);
  abstract T visit(Residue r);
}
class Natural extends Equalizable<Natural>{
  // ... snap
  <T>T accept(NaturalVisitor<T> v){
    return v.visit(this);
  }
  public boolean equals(Natural n){
    return n.accept(new NaturalVisitor<Boolean>(){
      Boolean visit(Residue r){ return r.number%r.mod==number; }
      Boolean visit(Natural n){ return n.number==number;}
    });
  }
}
```

✓ Works

Visitor-Pattern

```
abstract class NaturalVisitor<T> {
  abstract T visit(Natural n);
  abstract T visit(Residue r);
}
class Natural extends Equalizable<Natural>{
  // ... snap
  <T>T accept(NaturalVisitor<T> v){
    return v.visit(this);
  }
  public boolean equals(Natural n){
    return n.accept(new NaturalVisitor<Boolean>(){
      Boolean visit(Residue r){ return r.number%r.mod==number; }
      Boolean visit(Natural n){ return n.number==number;}
    });
  }
}
```

✓ Works & Implementation of all possibilities is enforced!

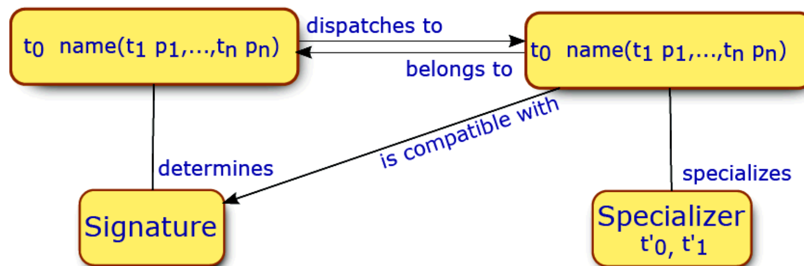
Visitor-Pattern

```
abstract class NaturalVisitor<T> {
  abstract T visit(Natural n);
  abstract T visit(Residue r);
}
class Natural extends Equalizable<Natural>{
  // ... snap
  <T>T accept(NaturalVisitor<T> v){
    return v.visit(this);
  }
  public boolean equals(Natural n){
    return n.accept(new NaturalVisitor<Boolean>(){
      Boolean visit(Residue r){ return r.number%r.mod==number; }
      Boolean visit(Natural n){ return n.number==number;}
    });
  }
}
```

✓ Works & Implementation of all possibilities is enforced!

⚠ But beyond Double-Dispatch?

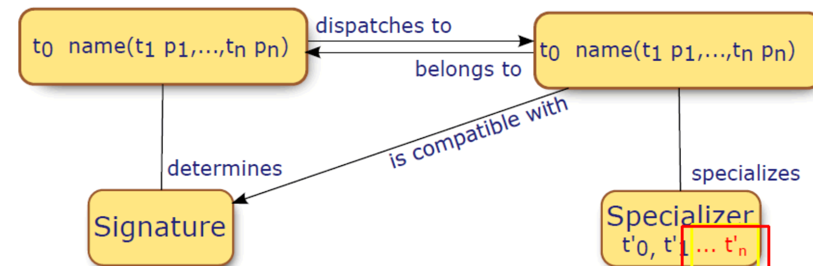
Formal Model of Multi-Dispatching [4]



Formal Model of Multi-Dispatching [4]

Idea

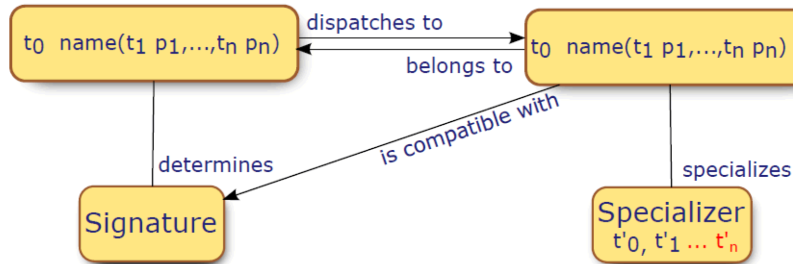
Introduce Specializers for all parameters



Formal Model of Multi-Dispatching [4]

Idea

Introduce Specializers for all parameters



How it works

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

Implications of the implementation

Type-Checking

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

Code-Generation

1. Specialized methods generated separately
2. Dispatcher method calls specialized methods
3. 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 in Multi-Java [1]

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


Natural Numbers in Multi-Java [1]

```
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);
```


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

✓ Clean Code!

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
```


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
```


⇝ Redirection to methods `equals$body3$1` and `equals$body3$0`

Natural Numbers in Clojure

```
(derive ::number ::class)

(defmulti equ (fn [a b] [a b]))
(defmethod equ [::number ::number] [a b] (pr "number-equals"))
(defmethod equ [::class  ::class]  [a b] (pr "ordinary equals"))

(equ ::number ::number)
(println)
```


Natural Numbers in Clojure

```
(derive ::number ::class)

(defmulti equ (fn [a b] [a b]))
(defmethod equ [::number ::number] [a b] (pr "number-equals"))
(defmethod equ [::class  ::class]  [a b] (pr "ordinary equals"))

(equ ::number ::number)
(println)
```


```
number-number
```

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. "Vivien" 500)))
(pr (print (UniPerson. "Seidl" 6000)))
```


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. "Vivien" 500)))
(pr (print (UniPerson. "Seidl" 6000)))
```



```
Dr. Simon
HiWi Vivien
Prof. Seidl
```


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 (MD)
- Scala (Dependent types, Mixin-Composition, etc.)

Lessons Learned

Lessons Learned

1. Dynamically dispatched methods are complex interaction of static and dynamic techniques
2. Single Dispatching as in major OO-Languages
3. Multi Dispatching generalizes single dispatching
4. Multi Dispatching Java
5. Multi Dispatching Clojure