

## Re: Is anything easier to do in java than in lisp?

**Source:** <http://coding.derkeiler.com/Archive/Lisp/comp.lang.lisp/2004-05/0674.html>

---

**From:** Antonio Menezes Leitao (*Antonio.Leitao\_at\_evaluator.pt*)

**Date:** 05/09/04

Date: Sun, 09 May 2004 10:40:16 +0100

On Sat, 08 May 2004 19:38:34 -0800, RobertMaas wrote:

- > *Static methods in java are much like ordinary functions in lisp, except*
- > *they don't have keyword arguments which means you must look up a host*
- > *of almost-the-same functions instead of just one function with a bunch*
- > *of keywords you can mix in any form. But most of the API has instance*
- > *methods instead of static methods, so for example if you want to find*
- > *the index where str1 occurs within str2, you're forced to re-write*
- > *(setq index (search str1 str2))*
- > *not as*
- > *index = String.indexOf(str1, str2)*
- > *but as*
- > *index = str2.indexOf(str1)*
- > *gee, it's even backwards from CL convention there.*

Unless someone does the re-write for you :-)

In (the next version of) Linj you write:

```
(setq index (search str1 str2))
```

and you get

```
index = str2.indexOf(str1)
```

- > *Anyway, back to lack of keywords, so every combination of what would be*
- > *keywords in CL becomes a totally separate functionName and/or*
- > *argumentList in java. For example:*
- > *int String.indexOf(int ch)*
- > *int String.indexOf(int ch, int fromIndex)*
- > *int String.lastIndexOf(int ch)*
- > *int String.lastIndexOf(int ch, int fromIndex)*
- > *whereas in CL you have a single function:*
- > *(position item sequence &key :from-end :test :test-not :start :end :key)*
- > *which not only works on strings, with :from-end making the difference*
- > *between indexOf and lastIndexOf, but :test :test-not and :key aren't*
- > *even available in java, and this same function works on all kinds of*
- > *sequences, not just strings, but other vectors, and linked-lists, too.*

## comp.lang.lisp: Re: Is anything easier to do in java than in lisp?

In (the next version of) Linj you write:

```
(position c str)
(position c str :from-end t)
(position c str :start 3)
(position c str :end 5 :from-end t)
```

and you get

```
str.indexOf(c)
str.lastIndexOf(c)
str.indexOf(c, 3)
str.lastIndexOf(c, 4)
```

but it's not a real replacement for Common Lisp. In the Linj version of this function you don't have `:test`, `:test-not` or `:key` and you can't mix `:start` and `(:from-end t)` or `:end` and `(:from-end nil)`. Compared with Common Lisp sequence operations, the Java String class is really poor. And if you need modifiable strings, you must use `StringBuffer` which is even worse.

> *In Java if you want to do this same thing with vectors or linked-lists,*  
> *you probably need to write the function yourself, because as far as I*  
> *can tell java.lang doesn't have a class for vectors nor for linked*  
> *lists, and I don't know where else to find something like Vector.indexOf*  
> *or LinkedList.indexOf etc.*

If you are talking about `java.util.Vector`, then Java does implement similar operations (and I included them in Linj). Regarding `LinkedList`, there is a subset of the operations, but `LinkedList`, IMHO, is not what Lispers want. Linj provides its own implementation (named `cons`) that is much more useful. Here is an example where all three data types are used with the same operation:

```
(defun test-string-list (str vect list)
  (declare (string str) (Vector vect) (cons list))
  (+ (position #\c str)
     (position #\c vect :from-end t)
     (position #\c list :key #'car :test #'eql)))
```

The translation you get in Java is:

```
public static int testStringList(String str, Vector vect, Cons list) {
    return str.indexOf('c') +
           vect.lastIndexOf(new Character('c')) +
           list.position(new Character('c'), Predicate2.EQL_FUNCTION, Cons.CAR_FUNCTION);
}
```

Also note that Linj had to adapt the primitive type argument whenever the method parameter required reference types.

## comp.lang.lisp: Re: Is anything easier to do in java than in lisp?

- > *LISP also has:*
- > *(search sequence1 sequence2 &key :from-end :test :test-not :key*
- > *:start1 :end1 :start2 :end2)*
- > *for which java implementes only a small portion of the cases as:*
- > *int String.indexOf(String str)*
- > *int String.indexOf(String str, int fromIndex) int*
- > *String.lastIndexOf(String str)*
- > *int String.lastIndexOf(String str, int fromIndex)*

Linj now supports (a subset of) search.

- > *Suppose you want to find the first or last occurrence of some name,*
- > *ignoring case. In CL it's trivial:*
- > *(search "robert" "Hi, this is Robert Maas here" :test #'char-equal) how*
- > *would you do that in java except by writing your own nested loop from*
- > *scratch?? How come the java API doesn't already include this??*

My guess: because they were targeting C++ programmers. Is there a standard C++ function that does that?

- >> *2. It makes working with lists easy.*
- >
- > *Indeed, in java working with linked lists must be an awful pain.*

It is!

- > *Either*
- > *your list can contain only one kind of element, so you declair your own*
- > *class to include link-cells whose data pointer is of that type, or you*
- > *use the generic Object type and deal with having to write code that*
- > *explicitly checks the case of every element at runtime.*

Precisely. But in some cases you can do type inference and insert casts automatically. Here is one (crazy) example:

```
(defun test-list ()
  (let ((list '(1 "2" 3)))
    (if (string= (second list) "two")
        (+ (first list) (third list))
        (first list))))
```

In Linj, '+' and 'string=' are operations which require certain types of arguments. You can't use them in any other way. So, if the compiler knows the actual argument types, it will check if they match its expectations. If it doesn't know the actual argument types (meaning that all it knows is that they are subclasses of Object, then it can restrict those types to what it expects, leaving the downcast typecheck to runtime, as can be seen in the following translation:

```
public static Object testList() {
  Cons list = Cons.list(Bignum.valueOf(1), "2", Bignum.valueOf(3));
```

Re: Is anything easier to do in java than in lisp?

comp.lang.lisp: Re: Is anything easier to do in java than in lisp?

```
if (((String)list.second()).equals("two")) {
    return ((Bignum)list.first()).add((Bignum)list.third());
} else {
    return list.first();
}
}
```

> *And I just noticed: The arguent to indexOf isn't a character at all,  
> it's an integer!! At least in CL you can directly pass a character  
> object as argument to function that searches for that character within a  
> string, instead of coercing it to an integer first!! In CL, if you pass  
> an integer instead of a character, it looks for that integer, not the  
> character with that ASCII code, for example: (position 65 '#\A #\B 65  
> 66))  
> will find the 65 instead of the #\A, returning 2 instead of 0 as the  
> index where it found that number.*

Humm. Strings are not lists of characters, neither in Common Lisp, nor in Java. The Java position is defendable on the argument that an integer can be used as a carcter for the indexOf operation without confusion.

However, it is interesting to note that your example works as expected in Linj:

```
(position 65 '#\A #\B 65 66))
```

is translated into

```
Cons.list(new Character('A'),
          new Character('B'),
          Bignum.valueOf(65),
          Bignum.valueOf(66)).
          positionKey(Bignum.valueOf(65), null, null, 1);
```

and the correct result (2) is computed.

Antonio Leitao.

PS: These examples will only work in the next Linj version. While testing them I found that the current Linj version lacked lots of useful stuff.