CSCI 3055 - Programming Languages, Assignment 2

Luisa Rojas-Garcia (100518772)

---

(4) Complete the following table:

	Advantage	Disadvantage
`(defn render [data] ...)`	• Easiest to develop.	• Not polymorphic. • Often disorganized, polluted and repetitive code.
Multimethod	• Most powerful. • Very flexible. • Able to dispatch based on any function of the method's arguments. • Can always add new methods using namespaces.	• Not great when definiting a **group** of polymorphic methods - the solution can get very messy. • Very expensive. • Slower than protocols.
Protocol	• Good for defining a group of polymorphic methods. • Dynamic. • Defines no new types or classes • Provide efficient polymorphic dispatch based on the type of the first argument provided • Ability to use some efficient JVM features.	• Does not support interfaces with variable argument lists such as `[this & args]`

(5) What are some ways of handling inheritance?

In most programming languages, polymorphism is tied to inheritance; however, the concept of concrete inheritance is not built into Clojure. Nonetheless, the language provides some methods

for achieving it without using concrete inheritance.

- **Multimethods** (`defmulti`): This facilty is a combination of a dispatch function and one or multiple other methods, with a dispatch value each. Said dispatching method is called first and then it's return value is matched to the corresponding method to be used.

  It's important to note that multimethods use the `isa?` function in place of the `=` one in order to match dispatch values to the correct method. This is relevant because it yields *hierarchies*.

  Multimethods allow to dispatch on multiple arguments as well as hierarchical dispatching.

- **Protocols** (`defprotocol`): Protocols provide functions in a namespace and are rather similar to interfaces in object oriented programming languages.

  In the same way interfaces are used in conjunction with classes in Java, protocols can be used in conjunction with deftype or defrecord. This way, objects can be mimicked.

  For protocols, Clojure allows the implementation of a protocol to be specified as a map that maps function names to implementations.

- **Records** (`defrecord`): Records are custom, map-like data types; they associate keys and values and one can look up said values in the same way one could with a map. The difference is that, in Clojure, the fields for the records can be specified and they can be extended to implement protocols.

  Fields are slots for data and using them is like specifying which keys a data structure should have. Note that instances are created the same way Java objects are: using the class instantiation *interop* (ability to interact with native Java constructs within Clojure) call.

- **Functions as parameters**: This is the simplest form of polymorphism in Clojure, since it provides multiple solutions with one interface - the parameter(s) provided.

  The function taking in the parameter, another function, does not have to have any knowledge about the structure of the data that is returned to the user.

  Given that the function being passed as parameter is polymorphic, then the data can be rendered into any format the user requests - so if the user requests a different format, then the function as parameter should be changed to something else, but not the "main" one.

(6) What does the following code do?

```
1 (defn g
2   ([f & colls]
3       (apply concat (apply map f colls))))
```

Line 1

Defines a function `g`.

Line 2

Defines arguments `f` and `colls`, with `f` being the *positional-param* and `colls` being the *rest-parameter*. This means that when the function is called with arguments that exceed the positional parameters, it will find the rest in a *seq* contained in the *rest parameter*. Otherwise, the `rest-param` will be `nil`.

Line 3

The innermost portion of the code, `map f colls`, lazily evaluates `f` on each element in the collection, with `colls` being x1, x2, x3, ... return f(x1), f(x2), f(x3), ... The `apply` function added at the beginning of this portion of code will apply the `map` function to the list formed by prepending intervening arguments to `colls`.

The outer portion, `concat`, will return the concatenation of the elements outputed previously by the `map` function. The `apply` function preceding it will allow it to apply `concat` to all the values returned in the first portion of code, in a similar way it was described above.

The way `apply` works, is first it builds up the arguments provided. Then, it builds up the function to be applied to the arguments; finally, `apply` invokes the function when ready.

Sources

https://blog.8thlight.com/myles-megyesi/2012/04/26/polymorphism-in-clojure.html

http://david-mcneil.com/post/1475458103/implementation-inheritance-in-clojure

https://clojuredocs.org/clojure.core/defprotocol

http://www.braveclojure.com/multimethods-records-protocols/

http://sisyphus.rocks/talks/protocols-multimethods-records#9

https://clojuredocs.org/clojure.core/map

http://clojure.org/reference/data_structures#Collections

http://csci3055u.github.io/3-clojure/01.intro.html

https://clojuredocs.org/clojure.core/concat

https://clojuredocs.org/clojure.core/apply