# Clojure implementation: class organization

ClojureCLR follows ClojureJVM very closely in its design, including the organization of interfaces and classes used in the implementation. Not counting the classes in the compiler, there are roughly 250 interfaces and classes defined in ClojureCLR. These range from relatively straightforward utilities such as BigDecimal and LineNumberingTextReader to the complicated, intertwined set of interfaces and classes that define the basic data structures manipulated by Clojure. A reasonably accurate and complete chart of those dependencies is shown below.

Diagram

Description automatically generated

And that leaves out a lot of standalone classes not involved in the big hierarchies.

So, you want to implement a new Clojure to match against ClojureJVM class for class? Just do a traversal of this graph. Then add the missing support classes. Then write a compiler.

What’s missing from the big graph, other than the compiler code? Read it and weep.

AtomicBoolean AtomicInteger AtomicLong AtomicReference<T> Box ClrArraySpec ClrTypeSpec EdnReader Closure JavaConcurrentDictionary<TKey, TValue> JReMatcher LazilyPersistentVector LispReader LockingTransaction Murmur3 Numbers RT Util WarnBoxedMathAttribute Clojure LineNumberingTextReader PushbackInputStream PushbackTextReader BindingHelpers ClojureBinder ClojureCreateInstanceBinder ClojureGetZeroArityMemberBinder ClojureInvokeMemberBinder ClojureOverloadResolverFactory ClojureOverloadResolver DynUtils IClojureBinder MetaAFn ClojureContext ClojureOps ClojureOptions ListGenericWrapper<T> DictionaryGenericWrapper<K, V> IEnumeratorOfTWrapper<T> IEnumerableOfTWrapper<T> Converter ImmutableDictionaryEnumerator Printf Properties Reflector

We can help make some sense of the others by segmenting the graph above into some functional groupings. The breakdowns I chose were roughly:

* IFn functionality –implementing the function interface
* IMeta functionality – providing metadata attached to various objects
* Reference-holding functionality – objects that hold a value
* IReduce functionality – implementing the reduce protocol,
* ISeq functionality – sequences galore. I have to break this down into two pictures, one on basic ISeq related functionality, another for specialized functionality for counting, indexing and related operations applicable to certain sequences
* Transient collections – support for transient and editable collections
* Definitions – support for things such as deftype, defrecord, and proxy

## IFn functionality

IFn is the interface implemented by any object that can be invoked as a Clojure function. IFn declares overloads for the invoke method of many arities. AFn provides an abstract class implementing all the invoke methods to throw a not-implemented exception. AFunction adds an IComparer.Compare method that allows an IFn to serve as a comparator (using its two-arg invoke). RestFn provides implementations of the invokes that return null. It serves as a base for function implementations that have a &rest arg.

The data structures implement IFn (via AFn, AFunction, RestFn, typically) so that they can appear in the functional position of a form being evaluated. For example, a hash map hm appearing in (hm x y) will cause the two-arg invoke to be called, which is wired to the valAt method of the map to do a lookup by key with a default value.

Diagram

Description automatically generated

## IMeta and IReference functionality

IMeta is implemented by objects that have attached metadata. It declares meta() for retrieving that metadata. IObj declares a method withMeta for attaching metadata to an object. Because most data structures are immutable, the implementation of withMeta typically creates a new object with identical value to the original, but with new metadata attached. IReference allows metadata to be altered or set ; this is not designed to respect immutability. The limited number of classes that base on AReference all have side-effecting operations associated with them anyway. Note that ARef is an abstract class that actually adds IDRef and IRef functionality to AReference for those classes that support all these interfaces (IMeta, IReference, IRef, and IDRef). We cover the latter two in the next section.

Diagram

Description automatically generated

## Reference objects

Some objects hold references to other objects. An Atom provides access to an object that is thread-safe, for example. Interface IDeref is for accessing (derefencing) the held value. IRef adds validators and watchers for changes. ARef is an abstract class providing a basic implementation of IRef. IBlockingDeref is IDeref with a timeout value. IPending allows you to check if a possibly delayed value is now available.

You will see IReference and AReference here. These have nothing to do with IDeref/IRef. They are covered in the previous section. It’s just that ARef provides implementations for these also, and the main players here – Atom, Agent, Ref, and Var – implement both sets of functionality.

Diagram

Description automatically generated

## Sequence functionality

There is so much functionality around sequences that I had to break this into two diagrams to help cope with the complexity.

Sequable is for any object that provide an ISeq. The base for most of the immutable, persistent, sequential data structures is IPersistentCollection, providing count/cons/empty/equiv methods.

We have: interfaces for more specialized collection types such as IPersistentMap and IPersistentVector; abstract classes which provide partial implementations such as APersistentSet and APersistentMap; and fully realized implementations such as PersistentList and PersistentHashMap.

Diagram

Description automatically generated

Generally, when wanting to manipulate a sequential data structure, i.e. a Sequable, you call its Sequable.seq() method to get an ISeq object that provides first/next/more/cons for iteration and adding an element, the minimum required functionality. Note that we have the interface inheritance chain: Sequable 🡨 IPersistentCollection 🡨 ISeq. That provides a basic functionality of

ISeq seq();  
int count();  
IPersistentCollection cons(object o);  
IPersistentCollection empty();  
bool equiv(object o);  
object first();  
ISeq next();  
ISeq more();  
new ISeq cons(object o);  
(Note the overload of cons introduced by ISeq. That’s a major pain in my CLR butt.)

By one means or another, every concrete type in that chart implement all three interfaces.

There is one mystery interface in the picture: Sequential. You have to look very closely to figure out which data structures fail to implement Sequential. I’ll save you some time: because IPersistentSet and IPersistentMap do not inherit from Sequential, their implementations all fail to do so: PersistentHashSet, PersistentTreeSet, PersistentArrayMap, PersistentHashMap, PersistentStructMap and PersistentTreeMap.

What is the practical import of this? One has to look at where Sequential appears in the code. Most occurrences are Equals/doEquals/equiv/doEquiv code and for very select set of classes.

APersistentVector.doEquals and APersistentVector.doEquiv:

// handles IPersistentVector and IList separately first  
if (!(obj is Sequential))  
 return false;  
 ms = RT.seq(obj);

ASeq.Equals and ASeq.equiv:

if (!(obj is Sequential || obj is IList))   
 return false;  
ISeq ms = RT.seq(obj);

LazySeq.Equals and LazySeq.equiv:

ISeq s = seq();  
if (s != null)  
 return s.Equals(obj); // or s.equiv(obj)  
else  
 return (obj is Sequential || obj is IList)   
 && RT.seq(obj) == null;

PersistentList.EmptyList.Equals:

return (obj is Sequential || obj is IList)   
 && RT.seq(obj) == null;

PersistentQueue.Equals and PersistentQueue.equiv:

if (!(obj is Sequential))  
 return false;  
ISeq ms = RT.seq(obj);

Diagram

Description automatically generated

Diagram

Description automatically generatedDiagram

Description automatically generated

Diagram

Description automatically generated