Clojure is built on abstractions: sequences, references, macros, and so forth. However, most of those abstractions are implemented in Java, as classes and interfaces. It is difficult to add new abstractions to the language (for example, a queue data structure) without implementing them in Java.

Clojure 1.2 introduces several new features to make it easier to implement new abstractions directly in Clojure, while still taking full advantage of the performance optimizations in the Java platform. Datatypes and protocols are roughly analogous to Java’s classes and interfaces, but they are more flexible.

Note As of this writing, Clojure 1.2 has not yet been released. Although the concepts will remain the same, there may be minor changes in naming or syntax from what we describe in this chapter.

Protocols

A protocol is a set of methods. The protocol has a name and an optional documentation string. Each method has a name, one or more argument vectors, and an optional documentation string. That's it! There are no implementations, no actual code.

Protocols are created with defprotocol:

```
(defprotocol MyProtocol
  "This is my new protocol"
  (method-one [x] "This is the first method."
  (method-two ([x] [x y]) "The second method.")
)
```

If you were to execute this example in the namespace `my.code`, the following Vars would be created:

- `my.code/MyProtocol`: A protocol object.
- `my.code/method-one`: A function of one argument.
- `my.code/method-two`: A function of one or two arguments.

`method-one` and `method-two` are polymorphic functions, meaning they can have different implementations for different types of objects. You can call `method-one` or `method-two` immediately after `defprotocol`, but they will throw an exception because no implementations have been defined.
What is a protocol? It’s a contract, a set of capabilities. An object or a datatype (described in the next section) can declare that it supports a particular protocol, meaning that it has implementations for the methods in that protocol.

**Protocols As Interfaces**

Conceptually, a protocol is similar to a Java interface. In fact, `defprotocol` creates a Java interface with the same methods. You can AOT-compile the Clojure source file containing `defprotocol` and use the interface in Java code. The Java interface will be in a package matching the namespace in which the protocol was defined. The package, interface, and method names will be adjusted to obey Java naming rules, such as replacing hyphens with underscores. Each method in the interface will have one argument fewer than the protocol method: that argument is the `this` pointer in Java. The previous example would create an interface matching the following Java code:

```java
package my.code;

public interface MyProtocol {
    public Object method_one();
    public Object method_two(Object y);
}
```

There is one important difference between protocols and interfaces: protocols have no inheritance. You cannot create “subprotocols” like Java’s subinterfaces.

Protocols are also similar to “mix-in” facilities provided by languages such as Ruby, with another important difference: protocols have no implementation. As a result, protocols never conflict with one another, unlike mix-ins.

**Datatypes**

Although Clojure is not, strictly-speaking, an object-oriented language, sometimes it is tempting to think in object-oriented terms when dealing with the real world. Most applications have many “records” of the same “type” with similar “fields.”

Prior to Clojure 1.2, the standard way to handle records was to use maps. This worked, but did not permit any performance optimizations from reusing the same keys in many maps.

StructMaps were one solution, but they had several problems. StructMaps have a predefined set of keys, but no actual “type” that can be queried at runtime. They cannot be printed and read back as StructMaps. They cannot have primitive-typed fields, and they cannot match the performance of instance fields in plain old Java objects.

Clojure 1.2 introduces datatypes as a replacement for StructMaps. A *datatype* is a named record type, with a set of named fields that can implement protocols and interfaces. Datatypes are created with `defrecord`:

```clojure
(defrecord name [fields...])
```

For example, a datatype might store an employee record with two fields, name and room number:

```clojure
user> (defrecord Employee [name room])
```