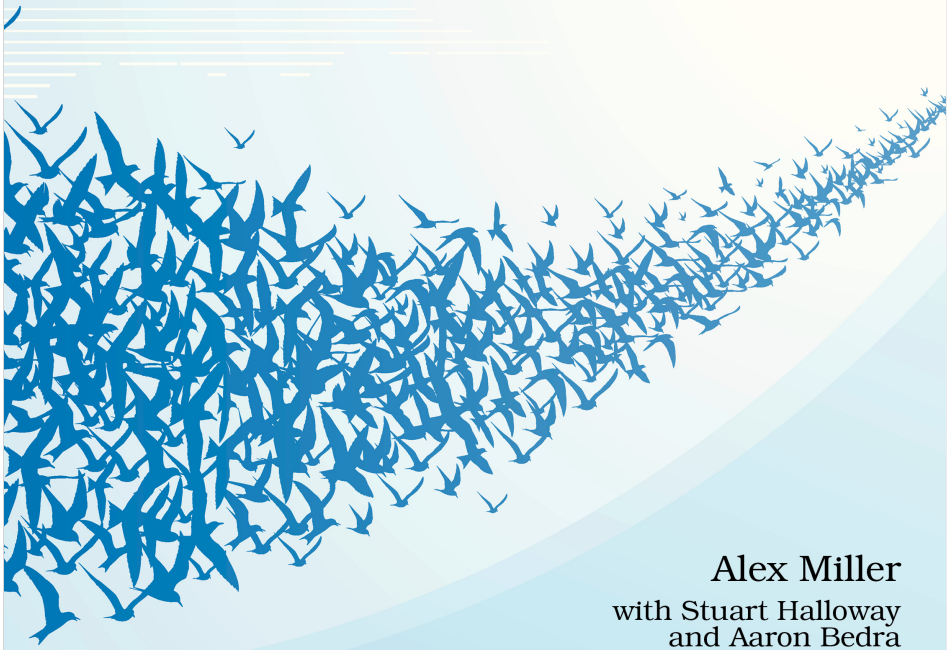
The Pragmatic Programmers

# Programming Clojure

## Fourth Edition



Alex Miller
with Stuart Halloway
and Aaron Bedra
*edited by Jacquelyn Carter*

This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.pragprog.com.

Copyright © The Pragmatic Programmers, LLC.

## Records

Classes in object-oriented programs tend to fall into two distinct categories: those that represent programming artifacts, such as String, Socket, InputStream, and OutputStream, and those that represent application domain information, such as Employee and PurchaseOrder.

Unfortunately, using classes to model application domain information hides it behind a class-specific micro-language of setters and getters. You can no longer take a generic approach to information processing, and you end up with a proliferation of unnecessary specificity and reduced reusability. See Clojure's documentation on datatypes[3] for more information.

For this reason, Clojure has always encouraged the use of maps for modeling such information, and that holds true even with datatypes, which is where records come in. A record is a datatype, like those created with deftype, that also implements PersistentMap and therefore can be used like any other map (mostly); and since records are also proper classes, they support type-based polymorphism through protocols. With records, we have the best of both worlds: maps that can implement protocols.

What could be more natural than using records to play music? So, let's create a record that represents a musical note, with fields for pitch, octave, and duration; then we'll use the JDK's built-in MIDI synthesizer to play sequences of these notes.

Since records are maps, we'll be able to change the properties of individual notes using the assoc and update-in functions, and we can create or transform entire sequences of notes using map and reduce. This gives us access to the entirety of Clojure's collection API.

All examples in the rest of this chapter will be evaluated in terms of this namespace definition:

src/examples/note.clj

```
(ns examples.note
  (:import [javax.sound.midi MidiSystem]))
```

We'll create a Note record with the defrecord macro, which behaves like deftype.

```
(defrecord name [& fields] & opts+specs)
```

3. https://clojure.org/reference/datatypes

A Note record has three fields: pitch, octave, and duration. All of the remaining examples in this chapter will be evaluated in the context of:

src/examples/note.clj
```
(defrecord Note [pitch octave duration])
```

The pitch will be represented by a keyword like :C, :C#, and :Db, which represent the notes C, C♯ (C sharp), and D♭ (D flat), respectively. Each pitch can be played at different octaves; for instance, middle C is in the fourth octave. Duration indicates the note length; a whole note is represented by 1, a half note by 1/2, a quarter note by 1/4, and a 16th note by 1/16. For example, we can represent a D♯ half note in the fourth octave with this Note record:

```
(->Note :D# 4 1/2)
-> #examples.note.Note{:pitch :D#, :octave 4, :duration 1/2}
```

Records *are* maps:

```
(map? (->Note :D# 4 1/2))
-> true
```

so we can also access their fields using keywords:

```
(:pitch (->Note :D# 4 1/2))
-> :D#
```

We can create modified records with assoc and update-in.

```
(assoc (->Note :D# 4 1/2) :pitch :Db :duration 1/4)
-> #examples.note.Note{:pitch :Db, :octave 4, :duration 1/4}
```

```
(update-in (->Note :D# 4 1/2) [:octave] inc)
-> #examples.note.Note{:pitch :D#, :octave 5, :duration 1/2}
```

Records are open, so we can associate extra fields into a record:

```
(assoc (->Note :D# 4 1/2) :velocity 100)
-> #examples.note.Note{:pitch :D#, :octave 4, :duration 1/2, :velocity 100}
```

Use the optional :velocity field to represent the force with which a note is played.

When used on a record, both assoc and update-in return a new record, but the dissoc function works a bit differently; it will return a new record if the field being dissociated is optional, like velocity in the previous example, but it will return a plain map if the field is mandated by the defrecord specification, like pitch, octave, or duration.

In other words, if you remove a required field from a record of a given type, it's no longer a record of that type, and it simply becomes a map.

```
(dissoc (->Note :D# 4 1/2) :octave)
-> {:pitch :D#, :duration 1/2}
```

• Click HERE to purchase this book now. discuss

Notice that dissoc returns a map, not a record. One difference between records and maps is that, unlike maps, records are not functions of keywords.

```
((->Note :D# 4 1/2) :pitch)
-> Execution error (ClassCastException) at user/eval176 (REPL:1).
-> class examples.note.Note cannot be cast to class clojure.lang.IFn
```

ClassCastException is thrown because records do not implement the IFn interface like maps do. Records are not simply functions of key to value, they have additional functionality, and may also implement the IFn interface to do something else.

When accessing a collection, you should place the collection first. When accessing a map that's acting (conceptually) as a data record, you should place the keyword first, even if the record is implemented as a plain map. Now that we have our basic Note record, let's add some methods so we can play them with the JDK's built-in MIDI synthesizer. We'll start by creating a MidiNote protocol with three methods:

src/examples/note.clj

```
(defprotocol MidiNote
  :extend-via-metadata true

  (msec [this tempo] "Note duration in ms")
  (key-number [this] "MIDI key number")
  (velocity [this]   "MIDI velocity"))
```

The :extend-via-metadata option is a protocol feature that we'll talk more about later in Metadata Extension, on page ?.

To play our note with the MIDI synthesizer, we need a representation of a MIDI note to play, which we capture in the MidiNote protocol. Given a note, we need to translate its pitch and octave into a MIDI key number and its duration into milliseconds. Also the note will be played with some velocity (usually treated as volume).

Before we extend the Note record to implement the MidiNote protocol, let's create a few helper methods. Creating these smaller utility functions makes them easier to test and the overall code easier to understand.

The beat->msec function translates the note's number of beats (whole note = 1, half note = 1/2, and so on), into milliseconds based on the given tempo, which is represented in beats per minute (bpm).

src/examples/note.clj

```
;; Assumes whole note = 4 beats
(defn beats->msec [beats tempo]
  (* (/ beats tempo) 4 60 1000))
```

The note->key function maps the keywords used to represent pitch into a number ranging from 0 to 11 and then uses this number along with the given octave to find the corresponding MIDI key-number.

src/examples/note.clj
```
(defn note->key [pitch octave]
  (let [scale {:C 0,  :C# 1, :Db 1,  :D 2,
               :D# 3, :Eb 3, :E 4,   :F 5,
               :F# 6, :Gb 6, :G 7,   :G# 8,
               :Ab 8, :A 9,  :A# 10, :Bb 10,
               :B 11}]
    (+ (* 12 (inc octave)) (scale pitch))))
```

With that in place, we can now extend Note to MidiNote.

src/examples/note.clj
```
(extend-type Note
  MidiNote
  (msec [this tempo]
    (beats->msec (:duration this) tempo))
  (key-number [this]
    (note->key (:pitch this) (:octave this)))
  (velocity [this]
    (or (:velocity this) 64)))
```

Now we need a function that sets up the MIDI synthesizer and plays a sequence of notes:

src/examples/note.clj
```
(defn perform [notes & {:keys [tempo] :or {tempo 120}}]
  (with-open [synth (doto (MidiSystem/getSynthesizer) .open)]
    (let [channel (aget (.getChannels synth) 0)]
      (doseq [note notes]
        (.noteOn channel (key-number note) (velocity note))
        (Thread/sleep (long (msec note tempo)))))))
```

The perform function takes a sequence of notes and an optional tempo value, opens a MIDI synthesizer, gets a channel from it, and then plays the note for the specified MIDI pitch and velocity. The note continues to play while the current thread is asleep, stopping only when the next note is sent to the channel.

All the pieces are in place, so let's make music using a sequence of Note records:

```
(def close-encounters [(->Note :D 3 1/2)
                       (->Note :E 3 1/2)
                       (->Note :C 3 1/2)
                       (->Note :C 2 1/2)
                       (->Note :G 2 1/2)])
```

```
-> #'user/close-encounters
```

In this case, our "music" consists of the five notes used to greet the alien ships in the movie *Close Encounters of the Third Kind*. To play it, just pass the sequence to the perform function:

```
(perform close-encounters)
-> nil
```

We can also generate sequences of notes dynamically with the for macro.

```
(def jaws (for [duration [1/2 1/2 1/4 1/4 1/8 1/8 1/8 1/8]
                pitch [:E :F]]
            (Note. pitch 2 duration)))
-> #'user/jaws

(perform jaws)
-> nil
```

The result is the shark theme from *Jaws*—a sequence of alternating E and F notes progressively speeding up as they move from half notes to quarter notes to eighth notes.

Since notes are records and records are map-like, we can manipulate them with any Clojure function that works on maps. For instance, we can map the update-in function across the *Close Encounters* sequence to raise or lower its octave.

```
(perform (map #(update-in % [:octave] inc) close-encounters))
-> nil

(perform (map #(update-in % [:octave] dec) close-encounters))
-> nil
```

Or we can create a sequence of notes that have progressively larger values of the optional :velocity field:

```
(perform (for [velocity [64 80 90 100 110 120]]
           (assoc (Note. :D 3 1/2) :velocity velocity)))
-> nil
```

This results in a sequence of increasingly more forceful D notes. Manipulating sequences is a particular strength of Clojure, so there are endless possibilities for programmatically creating and manipulating sequences of Note records.

## reify

The reify macro lets you create an anonymous instance of a datatype that implements either a protocol or an interface. You do not need to declare fields or methods; the object returned only has the methods of the interface or

protocol it implements. This is useful when you need a one-time implementation of an interface or protocol without a reusable type.

```
(reify & opts+specs)
```

reify, like deftype and defrecord, takes the name of one or more protocols, or interfaces, and a series of method bodies. Unlike deftype and defrecord, it doesn't take a name or a vector of fields; datatype instances produced with reify don't have explicit fields, relying instead on closures.

Let's compose some John Cage–style[4] aleatoric music[5] or, better yet, create an aleatoric music generator. We'll use reify to create an instance of a MidiNote that will play a different random note each time its play method is called.

src/examples/note.clj

```
(let [min-duration 250
      min-velocity 64
      rand-note (reify MidiNote
                  (msec [_ tempo] (+ (rand-int 1000) min-duration))
                  (key-number [_] (rand-int 100))
                  (velocity [_] (+ (rand-int 100) min-velocity)))]
  (perform (repeat 15 rand-note)))
```

First we bind two values, min-duration and min-velocity, that we will use in the MidiNote method implementations. Then we use reify to create an instance of an anonymous type, which implements the MidiNote protocol, that will select a random note, duration, and velocity each time its play method is called. Finally, we use the repeat function to create a sequence of 15 notes, consisting of a single instance of rand-note, and perform it. *Voila*, you now have a virtual John Cage!

4. https://en.wikipedia.org/wiki/John_Cage
5. https://en.wikipedia.org/wiki/Aleatoric_music