ClojureScript Unraveled

Andrey Antukh **<niwi@niwi.be>**

Alejandro Gomez **<alejandro@dialelo.com>**

Revision 1
2015-05-04

Table of Contents

Chapter 1. About this book

This book covers the ClojureScript programming language, a detailed guide of its tooling for development and a series of articles about topics that are applicable in day-to-day programming in ClojureScript.

It is not an introductory book to programming in that it assumes the reader has experience programming in at least one language. However, it doesn't assume experience with ClojureScript or functional programming. We'll try to include links to reference material when talking about the theoretical programming language aspects of ClojureScript that may be not be familiar to everybody.

Also, since the ClojureScript documentation is good but disperse, we wanted to write a compendium of reference information and extensive examples to serve as a ClojureScript primer as well as a series of practical how-to's. This document will evolve with the ClojureScript language, both as a reference of the language features and a sort of cookbook with practical programming recipes.

You'll get the most ouf this book if you:

- are curious about ClojureScript or functional programming and you have some programming experience;
- write JavaScript or any other language that compiles to it and want to know what ClojureScript has to offer;
- you already know some Clojure and want to learn how ClojureScript differs from it, plus practical topics like how to target both languages with the same code base.

Don't be turned off if you don't see yourself in neither of the above groups, we encourage you to give this book a try and give us feedback on how we can make it more accesible. Our goal is to make ClojureScript more friendly to newcomers and spread the ideas about programming that Clojure has helped popularize, as we see a lot of value in them.

Chapter 2. Introduction

This chapter is an introduction to the Clojure ecosystem, and intends to explain the philosophy behind it.

2.1. First contact

ClojureScript is an implementation of the Clojure programming language that targets JavaScript. Because of this it can run in many different execution environments including web browsers, Node.js, io.js, and Nashhorn.

Unlike other languages that intend to *compile* to JavaScript (like TypeScript, FunScript or CoffeeScript), ClojureScript is designed to use JavaScript like bytecode. It embraces functional programming, and has very safe and consistent defaults.

Another big difference (and in my opinion a big advantage) over other languages is that Clojure is designed to be a guest. It is a language without its own virtual machine that can be easy adapted to differences between its execution environments.

TBD

2.2. The pillars of the language

TBD

2.3. Why host on JavaScript?

TBD

Chapter 3. The language.

This chapter will be a little introduction to ClojureScript without assumptions about previous knowledge of the Clojure language, providing a quick tour over all the things you will need to know in order to understand the rest of this book.

3.1. First steps with Lisp syntax

Invented by John McCarthy in 1958, Lisp is one of the oldest programming languages that is still around. It has evolved into many derivatives called dialects, and ClojureScript is one of them. It's a programming language written in its own data structures, originally lists enclosed in parentheses, but Clojure(Script) has evolved the Lisp syntax with more data structures, making it more pleasant to write and read.

A list with a function in the first position is used for calling a function in ClojureScript:

```
(+ 1 2 3)
;; => 6
```

In the example above, we're applying the addition function **+** to the arguments 1, 2 and 3. ClojureScript allows many unusual characters like **?** or **-** in symbol names, which makes it easier to read:

```
(zero? 0)
;; => true
```

For distinguishing function calls and lists, we can quote lists to keep them from being evaluated. The quoted lists will be treated as data instead of as a function call:

```
'(+ 1 2 3)
;; => (+ 1 2 3)
```

ClojureScript uses more than lists for its syntax. The full details will be covered later, but here is an example of the usage of a vector (enclosed in brackets) for defining local bindings:

```
(let [x 1
      y 2
      z 3]
```

```
   (+ x y z))
;; => 6
```

This is practically all the syntax we need to know for using not only ClojureScript, but any Lisp. Being written in its own data structures (often referred to as *homoiconicity*) is a great property since the syntax is uniform and simple; also, code generation via macros is easier than in any other language, giving us plenty of power for extending the language to suit our needs.

3.2. The base data types

The ClojureScript language has a rich set of data types like most programming languages. It provides scalar datatypes that will be very familiar to you, such as numbers, strings, and floats. Beyond these, it also provides a great number of others that might be less familiar, such as symbols, keywords, regex (regular expressions), vars, atoms, volatiles…

ClojureScript embraces the host language, and where possible it uses the host's provided types. For example: numbers and strings are used as is and they behave in same way as in JavaScript.

3.2.1. Numbers

In *ClojureScript*, numbers include both integers and floating points. Keeping in mind that *ClojureScript* is a guest language that compiles to JavaScript, integers are actually JavaScript's native floating points under the hood.

As in any other language, numbers in *ClojureScript* are represented in following way:

```
23
+23
-100
1.7
-2
33e8
12e-14
3.2e-4
```

3.2.2. Keywords

Keywords in *ClojureScript* are objects that always evaluate to themselves. They are usually used in map data structures to efficiently represent the keys.

```
:foobar
:2
:?
:foo/bar
```

As you can see, the keywords are all prefixed with `:`, but this character is only part of the literal syntax and is not part of the name of the object.

You can also create a keyword by calling the **`keyword`** function. Don’t worry if you don’t understand or are unclear about anything in the following example; the functions are discussed in later chapters.

```
(keyword "foo")
;; => :foo
```

3.2.3. Symbols

Symbols in *ClojureScript* are very, very similar to **Keywords** (which you now know about). But instead of evaluating to themselves, symbols are evaluated to something that they refer to, that can be functions, variables, etc.

Symbols are represented with something that does not start with a number:

```
sample-symbol
othersymbol
f1
```

Don’t worry if you don’t understand right away; symbols are used in almost all of our examples, which will give you the opportunity to learn more as we go on.

3.2.4. Strings

There is almost nothing new we can explain about strings that you don’t already know. In *ClojureScript*, they work the same as in any other language. One point of interest, however, is that they are immutable.

In this case they are the same as in JavaScript:

```
"A example of a string"
```

One peculiar aspect of Strings in *ClojureScript* is due to the language's Lisp syntax: single and multiline strings have the same syntax:

```
"This is a multiline
      string in ClojureScript."
```

3.2.5. Characters

ClojureScript also lets you write single characters using Clojure's character literal syntax.

```
\a        ; The lowercase a character
\newline  ; The new line character
```

Since the host language doesn't contain character literals, *ClojureScript* characters are transformed behind the scenes into single character JavaScript strings.

3.2.6. Collections

Another big step in explaining a language is to explain its collections and collection abstractions. *ClojureScript* is not an exception to this rule.

ClojureScript comes with many types of different collections. The main difference between *ClojureScript* collections and collections in other languages is that they are persistent and immutable.

Before moving on to all of these (possibly) unknown concepts, we'll present a high level overview of existing collection types in *ClojureScript*.

Lists

This is a classic collection type in languages based on Lisp. Lists are the simplest type of collection in *ClojureScript*. Lists can contain items of any type, including other collections.

Lists in *ClojureScript* are represented by items enclosed between parentheses:

```
'(1 2 3 4 5)
'(:foo :bar 2)
```

As you can see, all list examples are prefixed with the `'` char. This is because lists in Lisp like languages are often used to express things like function or macro calls. In that case the first item should be a symbol that will evaluate to a something callable, and the rest of the list elements will be function parameters. However, in the preceding examples, we don't want the first item as a symbol; we just want a list of items. The following example shows the difference between a list without and with the preceding single quote mark:

```
(inc 1)
;; => 2

'(inc 1)
;; => (inc 1)
```

As you see, if you evaluate **`(inc 1)`** without prefixing it with **`'`**, it will resolve the **`inc`** symbol to the **inc** function and will execute it with **`1`** as first parameter returning the value **`2`**.

You can also explicitly create a list with the **`list`** function:

```
(list 1 2 3 4 5)
;; => (1 2 3 4 5)

(list :foo :bar 2)
;; => (:foo :bar 2)
```

Lists have the pecularity that they are very efficient if you access them sequentially or access their first elements, but a list is not a very good option if you need random (index) access to it's elements.

Vectors

Like lists, **Vectors** store a series of values, but in this case with very efficient index access to their elements, as opposed to lists, which are evaluated in order. Don't worry; in the following chapters we'll go in depth with details, but at this moment, this simple explanation is more than enough.

Vectors use square brackets for the literal syntax; let's see some examples:

```
[:foo :bar]
```

```
[3 4 5 nil]
```

Like lists, vectors can contain objects of any type, as you can observe in the preceding example.

You can also explicitly create a vector with the **`vector`** function, but this is not commonly used in ClojureScript programs:

```
(vector 1 2 3)
;; => [1 2 3]

(vector "blah" 3.5 nil)
;; => ["blah" 3.5 nil]
```

Maps

Maps are a collection abstraction that allows you to store key/value pairs. In other languages this type of structure is commonly known as a hash-map or dicts (dictionaries). Map literals in *ClojureScript* are written with the pairs between curly braces.

```
{:foo "bar", :baz 2}
{:alphabet [:a :b :c]}
```

Commas are frequently used to separate a key-value pair but are completely optional. In *ClojureScript* syntax, commas are treated like spaces.

Like Vectors, every item in a map literal is evaluated before the result is stored in a map, but the order of evaluation is not guaranteed.

Sets

And finally, **Sets**.

Sets store zero or more unique items of any type and are unordered. They, like maps, use curly braces for their literal syntax, with the difference being that they use a `#` as leading character:

```
#{1 2 3 :foo :bar}
```

In subsequent chapters we'll go in depth about sets and the other collection types you've seen in this chapter.

3.3. Vars

ClojureScript is a mostly functional language and focused on immutability. Because of that, it does not have the concept of variables as you know them in most other programming languages. The closest analogy to variables are the variables you define in algebra; when you say **`x = 6`** in mathematics, you are saying that you want the symbol **`x`** to stand for the number six.

In *ClojureScript*, vars are represented by symbols and store a single value together with metadata.

You can define a var using a **`def`** special form:

```
(def x 22)
(def y [1 2 3])
```

Vars are always top level in the namespace (which we will explain later). If you use **`def`** in a function call, the var will be defined at the namespace level, but we do not recommend this - instead, you should use **`let`** to define variables within a function.

3.4. Functions

3.4.1. The first contact

It's time to make things happen. *ClojureScript*, has what are known as first class functions. They behave like any other type, you can pass them as parameters and you can return them as values, always respecting the lexical scope. *ClojureScript* also has some features of dynamic scoping, but this will be discussed in other section.

If you want know more about scopes, this wikipedia article[1] is very extensive and explains different types of scoping.

As *ClojureScript* is a Lisp dialect, it uses the prefix notation for calling a function:

```
(inc 1)
;; => 2
```

[1] http://en.wikipedia.org/wiki/Scope_(computer_science)

In the example above, **`inc`** is a function and is part of the *ClojureScript* runtime, and **`1`** is the first argument for the **`inc`** function.

```
(+ 1 2 3)
;; => 6
```

The **`+`** symbol represents an **`add`** function. It allows multiple parameters, whereas in ALGOL-type languages, **`+`** is an operator and only allows two parameters.

The prefix notation has huge advantages, some of them not always obvious. *ClojureScript* does not make a distinction between a function and operator; everything is a function. The immediate advantage is that the prefix notation allows an arbitrary number of arguments per "operator". Also, it completely eliminates the problem of operator precedence.

3.4.2. Defining your own functions

You can define an un-named (anonymous) function with the **`fn`** special form. This is one type of function definition; in the following example, the function takes two parameters and returns their average.

```
(fn [param1 param2]
  (/ (+ param1 param2) 2.0))
```

You can define a function and call it at same time (in a single expression):

```
((fn [x] (* x x)) 5)
;; => 25
```

Let's start creating named functions. But what does a *named function* really mean? It is very simple; as in *ClojureScript*, functions are first-class and behave like any other value, so naming a function is done by simply binding the function to a symbol:

```
(def square (fn [x] (* x x)))

(square 12)
;; => 144
```

ClojureScript also offers the **`defn`** macro as a little syntactic sugar for making function definition more idiomatic:

```
(defn square
  "Return the square of a given number."
  [x]
  (* x x))
```

The string that comes between the function name and the parameter vector is called a *docstring* (documentation string); programs that automatically create web documentation from your source files will use these docstrings.

3.4.3. Function with multiple arities

ClojureScript also comes with the ability to define functions with arbitrary number of arguments. (The term *arity* means the number of arguments that a function takes.) The syntax is almost the same as for defining an ordinary function, with the difference that it has more than one body.

Let’s see an example, which will surely explain it much better:

```
(defn myinc
  "Self defined version of parameterized `inc`."
  ([x] (myinc x 1))
  ([x increment]
   (+ x increment)))
```

This line: **`([x] (myinc x 1))`** says that if there is only one argument, call the function **`myinc`** with that argument and the number **`1`** as the second argument. The other function body: **`([x increment] (+ x increment))`** says that if there are two arguments, return the result of adding them.

Here are some examples using the previously defined multi-arity function. Observe that if you call a function with wrong number of arguments, the compiler will emit an error message.

```
(myinc 1)
;; => 1

(myinc 1 3)
;; => 4

(myinc 1 3 3)
;; Compiler error
```

Explaining the concept of "arity" is out of the scope of this book, however you can read about that in this wikipedia article[2].

3.4.4. Variadic functions

Another way to accept multiple parameters is defining variadic functions. Variadic functions are functions that will be able to accept an arbitrary number of arguments:

```
(defn my-variadic-set
  [& params]
  (set params))

(my-variadic-set 1 2 3 1)
;; => #{1 2 3}
```

The way to denote a variadic function is using the **`&`** symbol prefix on its arguments vector.

3.4.5. Short syntax for anonymous functions

ClojureScript provides a shorter syntax for defining anonymous functions using the **`#()`** reader macro (usually leads to one liners). Reader macros are "special" expressions that will be transformed to the appropriate language form at compile time; in this case, to some expression that uses **`fn`** special form.

```
(def my-set #(set (list %1 %2)))

(my-set 1 2)
;; => #{1 2}
```

The preceding definition is shorthand for:

```
(def my-set-longer (fn [a b] #(set (list a b))))
```

The **`%1`**, **`%2`**, **`%N`** are simple markers for parameter positions that are implicitly declared when the reader macro will be interpreted and converted to a **`fn`** expression.

Also, if a function only accepts one argument, you can omit the number after **`%`** symbol; the function **`#(set (list %1))`** can be written **`#(set (list %))`**.

[2] http://en.wikipedia.org/wiki/Arity

Additionally, this syntax also supports the variadic form with the`%&` symbol:

```
(def my-variadic-set #(set %&))

(my-variadic-set 1 2 2)
;; => #{1 2}
```

3.5. Flow control

ClojureScript has a very different approach for flow control than languages like JavaScript, C, etc.

3.5.1. Branching with **`if`**

Let start with a basic one: **`if`**. In *ClojureScript* the **`if`** is an expression and not a statement, and it has three parameter: the first one is the condition expression, the second one is an expression that will be evaluated if the condition expression evaluates to logical true, and the third expression will be evaluated otherwise.

```
(defn discount
  "You get 5% discount for ordering 100 or more items"
  [quantity]
  (if (>= quantity 100)
    0.05
    0))

(discount 30)
;; => 0

(discount 130)
;; => 0.05
```

The block expression **`do`** can be used to have multiple expressions in an **`if`** branch. **`do`** is explained in the next section.

3.5.2. Branching with **`cond`**

Sometimes, the **`if`** expression can be slightly limited because it does not have the "else if" part to add more than one condition. The **`cond`** comes to the rescue.

With the **`cond`** expression, you can define multiple conditions:

```
(defn mypos?
  [x]
  (cond
    (> x 0) "positive"
    (< x 0) "negative"
    :else "zero"))

(mypos? 0)
;; => "zero"

(mypos? -2)
;; => "negative"
```

Also, **cond** has another form, called **condp**, that works very similarly to the simple **cond** but looks cleaner when the condition (also called a predicate) is the same for all conditions:

```
(defn translate-lang-code
  [code]
  (condp = (keyword code)
    :es "Spanish"
    :en "English"
    "Unknown"))

(translate-lang-code "en")
;; => "English"

(translate-lang-code "fr")
;; => "Unknown"
```

The line **`condp = (keyword code)`** means that, in each of the following lines, *ClojureScript* will apply the **=** function to the given keyword and the **code** argument.

3.5.3. Branching with `case`

The **case** branching expression has very similar use case as our previous example with **condp**. The main difference is that **case** always uses the **=** predicate/function, and its branching values are evaluated at compile time. This results in a more performant form than **cond** or **condp** but has the disadvantage that the condition value must be a static value.

Here is the same example as previous one, but using **case**:

```
(defn translate-lang-code
  [code]
  (case code
    "es" "Spanish"
    "es" "English"
    "Unknown"))

(translate-lang-code "en")
;; => "English"

(translate-lang-code "fr")
;; => "Unknown"
```

3.6. Locals, Blocks and Loops

3.6.1. Locals

ClojureScript does not has the concept of variables as in ALGOL-like languages, but it does have locals. Locals, as per usual, are immutable, and if you try mutate them, the compiler will throw an error.

The locals are defined with the **`let`** expression. The expression starts with a vector as first parameter followed by arbitrary number of expressions. The first parameter (the vector) should contain an arbitrary number of pairs that give a *binding form* (usually a symbol) followed by an expression whose value will be bound to this new local for the remainder of the let expression.

```
(let [x (inc 1)
      y (+ x 1)]
  (println "Simple message from the body of a let")
  (* x y))
;; Simple message from the body of a let
;; => 6
```

In the preceding example, the symbol **`x`** is bound to the value **`(inc 1)`**, which comes out to 2, and the symbol **`y`** is bound to the sum of **`x`** and 1, which comes out to 3. Given those bindings, the expressions **`(println "Simple message from the body of a let")`** and **`(* x y)`** are evaluated.

3.6.2. Blocks

In JavaScript, braces **{** and **}** delimit a block of code that “belongs together.” Blocks in *ClojureScript* are created using the **do** expression and are usually used for side effects, like printing something to the console or writing a log in a logger.

A side effect is something that is not necessary for the return value.

The **do** expression accepts as its parameter an arbitrary number of other expressions, but it returns the return value only from the last one:

```
(do
   (println "hello world")
   (println "hola mundo")
   (* 3 5) ;; this value will not be returned; it is thrown away
   (+ 1 2))

;; hello world
;; hola mundo
;; => 3
```

The body of the **let** expression, explained in previous section, is very similar to the **do** expression, in that it allows multiple expressions. In fact, the **let** has an implicit **do**.

3.6.3. Loops

The functional approach of *ClojureScript* means that it does not have standard, well known statement-based loops such as **for** in JavaScript. The loops in *ClojureScript* are handled using recursion. Recursion sometimes requires additional thinking about how to model your problem in a slightly different way than imperative languages.

Also, many of the common patterns for which **for** is used in other languages are achieved through higher-order functions - functions that accept other functions as parameters.

Looping with loop/recur

Let’s take a look at how to express loops using recursion with the **loop** and **recur** forms. **loop** defines a possibly empty list of bindings (notice the symmetry with **let**) and **recur** jumps execution back to the looping point with new values for those bindings.

Let’s see an example:

```
(loop [x 0]
   (println "Looping with " x)
   (if (= x 2)
     (println "Done looping!")
     (recur (inc x))))
;; Looping with 0
;; Looping with 1
;; Looping with 2
;; Done looping!
;; => nil
```

In the above snippet, we bind the name **`x`** to the value **`0`** and execute the body. Since the condition is not met the first time it’s run we **`recur`**, incrementing the binding value with the **`inc`** function. We do this once more until the condition is met and, since there aren’t more **`recur`** calls, exit the loop.

Note that **`loop`** isn’t the only point we can **`recur`** to; using **`recur`** inside a function executes the body of the function recursively with the new bindings:

```
(defn recursive-function [x]
   (println "Looping with" x)
   (if (= x 2)
     (println "Done looping!")
     (recur (inc x))))

(recursive-function 0)
;; Looping with 0
;; Looping with 1
;; Looping with 2
;; Done looping!
;; => nil
```

Replacing for loops with higher-order functions

In imperative programming languages it is common to use **`for`** loops to iterate over data and transform it, usually the intent being one of the following:

- Transform every value in the iterable yielding another iterable
- Filter the elements of the iterable by certain criteria

- Convert the iterable to a value where each iteration depends on the result from the previous one
- Run a computation for every value in the iterable

The above actions are encoded in higher-order functions and syntactic constructs in ClojureScript; let's see an example of the first three.

For transforming every value in an iterable data structure we use the **`map`** function, which takes a function and a sequence and applies the function to every element:

```
(map inc [0 1 2])
;; => (1 2 3)
```

For filtering the values of a data structure we use the **`filter`** function, which takes a predicate and a sequence and gives a new sequence with only the elements that returned **`true`** for the given predicate:

```
(filter odd? [1 2 3 4])
;; => (1 3)
```

Converting an iterable to a single value, accumulating the intermediate result at every step of the iteration can be achieved with **`reduce`**, which takes a function for accumulating values, an optional initial value and a collection:

```
(reduce + 0 [1 2 3 4])
;; => 10
```

`for` sequence comprehensions

In ClojureScript the **`for`** construct isn't used for iteration but for generating sequences, an operation also known as "sequence comprehension". It offers a small domain specific language for declaratively building sequences.

`for` takes a vector of bindings and a expression and generates a sequence of the result of evaluating the expression. Let's take a look at an example:

```
(for [x [1 2 3]]
  [x x])
;; => ([1 1] [2 2] [3 3])
```

In this example, **`x`** is bound to each of the items in the vector **`[1 2 3]`** in turn, and returns a new sequence of two-item vectors with the original item repeated.

`for` supports multiple bindings, which will cause the collections to be iterated in a nested fashion, much like nesting **`for`** loops in imperative languages. The innermost binding iterates “fastest.”

```
(for [x [1 2 3]
      y [4 5]]
  [x y])

;; => ([1 4] [1 5] [2 4] [2 5] [3 4] [3 5])
```

We can also follow the bindings with three modifiers: **`:let`** for creating local bindings, **`:while`** for breaking out of the sequence generation, and **`:when`** for filtering out values.

Here’s an example of local bindings using the **`:let`** modifier; note that the bindings defined with it will be available in the expression:

```
(for [x [1 2 3]
      y [4 5]
      :let [z (+ x y)]]
  z)
;; => (5 6 6 7 7 8)
```

We can use the **`:while`** modifier for expressing a condition that, when it is no longer met, will stop the sequence generation. Here’s an example:

```
(for [x [1 2 3]
      y [4 5]
      :while (= y 4)]
  [x y])

;; => ([1 4] [2 4] [3 4])
```

For filtering out generated values, use the **`:when`** modifier as in the following example:

```
(for [x [1 2 3]
      y [4 5]
      :when (= (+ x y) 6)]
  [x y])
```

```
;; => ([1 5] [2 4])
```

We can combine the modifiers shown above for expressing complex sequence generations or more clearly expressing the intent of our comprehension:

```
(for [x [1 2 3]
      y [4 5]
      :let [z (+ x y)]
      :when (= z 6)]
  [x y])

;; => ([1 5] [2 4])
```

When we outlined the most common usages of the **`for`** construct in imperative programming languages, we mentioned that sometimes we want to run a computation for every value in a sequence, not caring about the result. Presumably we do this for achieving some sort of side-effect with the values of the sequence.

ClojureScript provides the **`doseq`** construct, which is analogous to **`for`** but executes the expression, discards the resulting values, and returns **`nil`**.

```
(doseq [x [1 2 3]
        y [4 5]
       :let [z (+ x y)]]
  (println x "+" y "=" z))

;; 1 + 4 = 5
;; 1 + 5 = 6
;; 2 + 4 = 6
;; 2 + 5 = 7
;; 3 + 4 = 7
;; 3 + 5 = 8
;; => nil
```

3.7. Collection types

3.7.1. Immutable and persistent

We mentioned before that ClojureScript collections are persistent and immutable, but we didn’t explain what that meant.

An immutable data structure, as its name suggest, is a data structure that can not be changed. In-place updates are not allowed in immutable data structures.

A persistent data structure is a data structure that returns a new version of itself when transforming it, leaving the original unmodified. ClojureScript makes this memory and time efficient using an implementation technique called *structural sharing*, where most of the data shared between two versions of a value is not duplicated, and transformations of a value are implemented by copying the minimal amount of data required.

Let's see an example of appending values to a vector using the **`conj`** (for "conjoin") operation:

```
(let [xs [1 2 3]
      ys (conj xs 4)]
  (println "xs:" xs)
  (println "ys:" ys))

;; xs: [1 2 3]
;; ys: [1 2 3 4]
;; => nil
```

As you can see, we derived a new version of the **`xs`** vector appending an element to it and got a new vector **`ys`** with the element added. However, the **`xs`** vector remained unchanged, because it is immutable.

For illustrating the structural sharing of ClojureScript data structures, let's compare whether some parts of the old and new versions of a data structure are actually the same object with the **`identical?`** predicate. We'll use the list data type for this purpose:

```
(let [xs (list 1 2 3)
      ys (cons 0 xs)]
  (println "xs:" xs)
  (println "ys:" ys)
  (println "(rest ys):" (rest ys))
  (identical? xs (rest ys)))

;; xs: (1 2 3)
;; ys: (0 1 2 3)
;; (rest ys): (1 2 3)
;; => true
```

As you can see in the example, we used **`cons`** (construct) to prepend a value to the **`xs`** list and we got a new list **`ys`** with the element added. The **`rest`** of the **`ys`** list (all the values but the first) are the same object in memory as the **`xs`** list, thus **`xs`** and **`ys`** share structure.

3.7.2. The sequence abstraction

One of the central ClojureScript abstractions is the Sequence, which can be thought of as a list and can be derived from any of the collection types. It is persistent and immutable like all collection types, and many of the core ClojureScript functions return sequences.

The types that can be used to generate a sequence are called "seqables"; we can call **`seq`** on them and get a sequence back. Sequences support two basic operations: **`first`** and **`rest`**. They both call **`seq`** on the argument we provide them:

```
(first [1 2 3])
;; => 1

(rest [1 2 3])
;; => (2 3)
```

Calling **`seq`** on a seqable can yield different results if the seqable is empty or not. It will return **`nil`** when empty and a sequence otherwise:

```
(seq [])
;; => nil

(seq [1 2 3])
;; => (1 2 3)
```

`next` is a similar sequence operation to **`rest`**, but it differs from the latter in that it yields a **`nil`** value when called with a sequence with one or zero elements. Note that, when given one of the aforementioned sequences, the empty sequence returned by **`rest`** will evaluate as a boolean true whereas the **`nil`** value returned by **`next`** will evaluate as false (see the section on *truthiness* later in this chapter).

```
(rest [])
;; => ()
```

```
(next [])
;; => nil

(rest [1 2 3])
;; => (2 3)

(next [1 2 3])
;; => (2 3)
```

nil-punning

The above behaviour of **`seq`** when coupled with the falsey nature of **`nil`** in boolean contexts make it an idiom for checking the emptyness of a sequence in ClojureScript, which is often referred to as nil-punning.

```
(defn print-coll
  [coll]
  (when (seq coll)
    (println "Saw " (first coll))
    (recur (rest coll))))

(print-coll [1 2 3])
;; Saw 1
;; Saw 2
;; Saw 3
;; => nil

(print-coll #{1 2 3})
;; Saw 1
;; Saw 3
;; Saw 2
;; => nil
```

`nil` is also both a seqable and a sequence, and thus it supports all the functions we saw so far:

```
(seq nil)
;; => nil

(first nil)
;; => nil

(rest nil)
```

```
;; => ()
```

Functions that work on sequences

The ClojureScript core functions for transforming collections make sequences out of their arguments and are implemented in terms of the generic sequence operations we learned about in the preceding chapter. This makes them highly generic, since we can use them on any data type that is seqable. Let’s see how we can use **`map`** with a variety of seqables:

```
(map inc [1 2 3])
;; => (2 3 4)

(map inc #{1 2 3})
;; => (2 4 3)

(map count {:a 41 :b 40})
;; => (2 2)

(map inc '(1 2 3))
;; => (2 3 4)
```

As you may have noticed, functions that operate on sequences are safe to use with empty collections or even **`nil`** values since they don’t need to do anything but return an empty sequence when encountering such values.

```
(map inc [])
;; => ()

(map inc #{})
;; => ()

(map inc nil)
;; => ()
```

We already saw examples with the usual suspects like **`map`**, **`filter`** and **`reduce`**, but ClojureScript offers a plethora of generic sequence operations in its core namespace. Note that many of the operations we’ll learn about either work with seqables or are extensible to user defined types.

We can query a value to know whether it’s a collection type with the **`coll?`** predicate:

```
(coll? nil)
;; => false

(coll? [1 2 3])
;; => true

(coll? {:language "ClojureScript" :file-extension "cljs"})
;; => true

(coll? "ClojureScript")
;; => false
```

Similar predicates exist for checking if a value is sequence (**`seq?`**) or a seqable (**`seqable?`**):

```
(seq? nil)
;; => false
(seqable? nil)
;; => false

(seq? [])
;; => false
(seqable? [])
;; => true

(seq? #{1 2 3})
;; => false
(seqable? #{1 2 3})
;; => true

(seq? "ClojureScript")
;; => false
(seqable? "ClojureScript")
;; => false
```

For collections that can be counted in constant time, we can use the **`count`** operation:

```
(count nil)
;; => 0

(count [1 2 3])
;; => 3

(count {:language "ClojureScript" :file-extension "cljs"})
```

```
;; => 2

(count "ClojureScript")
;; => 13
```

We can also get an empty variant of a given collection with the **`empty`** function:

```
(empty nil)
;; => nil

(empty [1 2 3])
;; => []

(empty #{1 2 3})
;; => #{}
```

The **`empty?`** predicate returns true if the given collection is empty:

```
(empty? nil)
;; => true

(empty? [])
;; => true

(empty? #{1 2 3})
;; => false
```

The **`conj`** operation adds elements to collections and may add them in different "places" depending on the collection. It adds them where it is most performant for the collection type, but note that not every collection has a defined order.

We can pass as many elements we want to add to **`conj`**; let's see it in action:

```
(conj nil 42)
;; => (42)

(conj [1 2] 3)
;; => [1 2 3]

(conj [1 2] 3 4 5)
;; => [1 2 3 4 5]

(conj '(1 2) 0)
```

```
;; => (0 1 2)

(conj #{1 2 3} 4)
;; => #{1 3 2 4}

(conj {:language "ClojureScript"} [:file-extension "cljs"])
;; => {:language "ClojureScript", :file-extension "cljs"}
```

Laziness

Most of ClojureScript's sequence-returning functions generate lazy sequences instead of eagerly creating a whole new sequence. Lazy sequences generate their contents as they are requested, usually when iterating over them. Laziness ensures that we don't do more work than we need to and gives us the possibility of treating potentially infinite sequences as regular ones.

3.7.3. Collections in depth

Now that we're acquainted with ClojureScript's sequence abstraction and some of the generic sequence manipulating functions, it's time to dive into the concrete collection types and the operations they support.

Lists

In ClojureScript lists are mostly used as a data structure for grouping symbols together into programs. Unlike in other Lisps, many of the syntactic constructs of ClojureScript use data structures different from the list (vectors and maps). This makes code less uniform, but the gains in readability are well worth the price.

You can think of ClojureScript lists as singly linked lists, where each node contains a value and a pointer to the rest of the list. This makes natural (and fast!) to add items to the front of the list since adding to the end would require traversal of the entire list. The prepend operation is performed using the **cons** (construct) function.

```
(cons 0 (cons 1 (cons 2 ())))
;; => (0 1 2)
```

We used the literal **()** to represent the empty list. Since it doesn't contain any symbols, it is not treated as a function call. However, when using list literals that contain elements, we need to quote them to prevent ClojureScript from evaluating them as a function call:

```
(cons 0 '(1 2))
;; => (0 1 2)
```

Since the head is the position that has constant time addition in the list collection, the **`conj`** operation on lists naturally adds item in the front:

```
(conj '(1 2) 0)
;; => (0 1 2)
```

Lists and other ClojureScript data structures can be used as stacks using the **`peek`**, **`pop`**, and **`conj`** functions. Note that the top of the stack will be the "place" where **`conj`** adds elements to, making **`conj`** equivalent to the stack’s push operation. In the case of lists, **`conj`** adds elements to the front of the list, **`peek`** returns the first element of the list, and **`pop`** returns a list with all the elements but the first one.

Note that the two operations that return a stack (**`conj`** and **`pop`**) don’t change the type of the collection used for the stack.

```
(def list-stack '(0 1 2))

(peek list-stack)
;; => 0

(pop list-stack)
;; => (1 2)

(type (pop list-stack))
;; => cljs.core/List

(conj list-stack -1)
;; => (-1 0 1 2)

(type (conj list-stack -1))
;; => cljs.core/List
```

One thing that lists are not particularly good at is random indexed access. Since they are stored in a single linked list-like structure in memory, random access to a given index requires a linear traversal in order to either retrieve the requested item or throw an index out of bounds error. Non-indexed ordered collections like lazy sequences also suffer from this limitation.

Vectors

Vectors are one of the most common data structures in ClojureScript. They are used as a syntactic construct in many places where more traditional Lisps use lists, for example in function argument declarations and **`let`** bindings.

ClojureScript vectors have enclosing brackets **`[]`** in their syntax literals. They can be created with **`vector`** and from another collection with **`vec`**:

```
(vector? [0 1 2])
;; => true

(vector 0 1 2)
;; => [0 1 2]

(vec '(0 1 2))
;; => [0 1 2]
```

Vectors are, like lists, ordered collections of heterogeneous values. Unlike lists, vectors grow naturally from the tail, so the **`conj`** operation appends items to the end of a vector. Insertion on the end of a vector is effectively constant time:

```
(conj [0 1] 2)
;; => [0 1 2]
```

Another thing that differentiates lists and vectors is that vectors are indexed collections and as such support efficient random index access and non-destructive updates. We can use the familiar **`nth`** function to retrieve values given an index:

```
(nth [0 1 2] 0)
;; => 0
```

Since vectors associate sequential numeric keys (indexes) to values, we can treat them as an associative data structure. ClojureScript provides the **`assoc`** function that, given an associative data structure and a set of key-value pairs, yields a new data structure with the values corresponding to the keys modified. Indexes begin at zero for the first element in a vector.

```
(assoc ["cero" "uno" "two"] 2 "dos")
```

```
;; => ["cero" "uno" "dos"]
```

Note that we can only **`assoc`** to a key that is either contained in the vector already or if it's the last position in a vector:

```
(assoc ["cero" "uno" "dos"] 3 "tres")
;; => ["cero" "uno" "dos" "tres"]

(assoc ["cero" "uno" "dos"] 4 "cuatro")
;; Error: Index 4 out of bounds [0,3]
```

Perhaps surprisingly, associative data structures can also be used as functions. They are functions of their keys to the values they are associated with. In the case of vectors, if the given key is not present an exception is thrown:

```
(["cero" "uno" "dos"] 0)
;; => "cero"

(["cero" "uno" "dos"] 2)
;; => "dos"

(["cero" "uno" "dos"] 3)
;; Error: Not item 3 in vector of length 3
```

As with lists, vectors can be also used as stack with the **`peek`**, **`pop`** and **`conj`** functions. Note, however, that vectors grow from the opposite end of the collection as lists:

```
(def vector-stack [0 1 2])

(peek vector-stack)
;; => 2

(pop vector-stack)
;; => [0 1]

(type (pop vector-stack))
;; => cljs.core/PersistentVector

(conj vector-stack 3)
;; => [0 1 2 3]

(type (conj vector-stack 3))
```

```
;; => cljs.core/PersistentVector
```

The **map** and **filter** operations return lazy sequences, but as it is common to need a fully realized sequence after performing those operations, vector-returning counterparts of such functions are available as **mapv** and **filterv**. They have the advantages of being faster than building a vector from a lazy sequence and making your intent more explicit:

```
(map inc [0 1 2])
;; => (1 2 3)

(type (map inc [0 1 2]))
;; => cljs.core/LazySeq

(mapv inc [0 1 2])
;; => [1 2 3]

(type (mapv inc [0 1 2]))
;; => cljs.core/PersistentVector
```

Maps

Maps are ubiquitous in ClojureScript. Like vectors, they are also used as a syntactic construct for attaching metadata to vars. Any ClojureScript data structure can be used as a key in a map, although it's common to use keywords since can also be called as functions.

ClojureScript maps are written literally as key-value pairs enclosed in braces **{}**. Alternatively, they can be created with the **hash-map** function:

```
(map? {:name "Cirilla"})
;; => true

(hash-map :name "Cirilla")
;; => {:name "Cirilla"}

(hash-map :name "Cirilla" :surname "Fiona")
;; => {:name "Cirilla" :surname "Fiona"}
```

Since regular maps don't have a specific order, the **conj** operation just adds one or more key-value pairs to a map. **conj** for maps expects one or more sequences of key-value pairs as its last arguments:

```
(def ciri {:name "Cirilla"})

(conj ciri [:surname "Fiona"])
;; => {:name "Cirilla", :surname "Fiona"}

(conj ciri [:surname "Fiona"] [:occupation "Wizard"])
;; => {:name "Cirilla", :surname "Fiona", :occupation "Wizard"}
```

Maps associate keys to values and, as such, are an associative data structure. They support adding associations with **`assoc`** and, unlike vectors, removing them with **`dissoc`**. Let's explore these functions:

```
(assoc {:name "Cirilla"} :surname "Fiona")
;; => {:name "Cirilla", :surname "Fiona"}

(dissoc {:name "Cirilla"} :name)
;; => {}
```

Maps are also functions of their keys, returning the values related to the given key. Unlike vectors, they return **`nil`** if we supply a key that is not present in the map:

```
({:name "Cirilla"} :name)
;; => "Cirilla"

({:name "Cirilla"} :surname)
;; => nil
```

ClojureScript also offers sorted hash maps which behave like their unsorted versions but preserve order when iterating over them. We can create a sorted map with default ordering with **`sorted-map`**:

```
(def sm (sorted-map :c 2 :b 1 :a 0))
;; => {:a 0, :b 1, :c 2}

(keys sm)
;; => (:a :b :c)
```

If we need a custom ordering we can provide a comparator function to **`sorted-map-by`**, let's see an example inverting the value returned by the built-in **`compare`** function. Comparator functions take two elements to compare and returns -1 (less than), 0 (equal) or 1 (greater than):

```
(def reverse-compare (comp - compare))

(def sm (sorted-map-by reverse-compare :a 0 :b 1 :c 2))
;; => {:c 2, :b 1, :a 0}

(keys sm)
;; => (:c :b :a)
```

Sets

Sets in ClojureScript have literal syntax as values enclosed in **`#{}`** and they can be created with the **`set`** constructor. They are unordered collections of values without duplicates.

```
(set? #{\a \e \i \o \u})
;; => true

(set [1 1 2 3])
;; => #{1 2 3}
```

Set literals can not contain duplicate values. If you accidentaly write a set literal with duplicates an error will be thrown:

```
#{1 1 2 3}
;; clojure.lang.ExceptionInfo: Duplicate key: 1
```

There are many operations that can be performed with sets, although are located in the **`clojure.set`** namespace and thus need to be imported. You'll learn the details of namespacing later, for now you only need to know that we are loading a namespace called **`clojure.set`** and binding it to the **`s`** symbol.

```
(require '[clojure.set :as s])

(def danish-vowels #{\a \e \i \o \u \æ \ø \å})
;; => #{"a" "e" "å" "æ" "i" "o" "u" "ø"}

(def spanish-vowels #{\a \e \i \o \u})
;; => #{"a" "e" "i" "o" "u"}

(s/difference danish-vowels spanish-vowels)
;; => #{"å" "æ" "ø"}
```

```
(s/union danish-vowels spanish-vowels)
;; => #{"a" "e" "å" "æ" "i" "o" "u" "ø"}

(s/intersection danish-vowels spanish-vowels)
;; => #{"a" "e" "i" "o" "u"}
```

A nice property of immutable sets is that they can be nested, languages that have mutable sets can end up containing duplicate values but that can’t happen in ClojureScript. In fact, all ClojureScript data structures can be nested arbitrarily due to immutability.

Sets also support the generic **`conj`** operation, as every other collection does.

```
(def spanish-vowels #{\a \e \i \o \u})
;; => #{"a" "e" "i" "o" "u"}

(def danish-vowels (conj spanish-vowels \æ \ø \å))
;; => #{"a" "e" "i" "o" "u" "æ" "ø" "å"}

(conj #{1 2 3} 1)
;; => #{1 3 2}
```

They acts as read-only associative data that associates the values it contains to themselves. Since every value except **`nil`** and **`false`** are falsy in ClojureScript, we can use sets as predicate functions:

```
(def vowels #{\a \e \i \o \u})
;; => #{"a" "e" "i" "o" "u"}

(get vowels \b)
;; => nil

(contains? vowels \b)
;; => false

(vowels \a)
;; => "a"

(filter vowels "Hound dog")
;; => ("o" "u" "o")
```

Sets have a sorted counterpart like maps do, created using the functions **`sorted-set`** and **`sorted-set-by`** which are analogous to map's **`sorted-map`** and **`sorted-map-by`**.

```
(def unordered-set #{[0] [1] [2]})
;; => #{[0] [2] [1]}

(seq unordered-set)
;; => ([0] [2] [1])

(def ordered-set (sorted-set [0] [1] [2]))
;; =># {[0] [1] [2]}

(seq unordered-set)
;; => ([0] [1] [2])
```

Queues

ClojureScript also provides a persistent and immutable queue. Queues don't have literal syntax since they are not used as pervasively as other collection types.

There are no convenient constructor functions for creating persistent queues. Instead of that, we can get an empty instance using **`PersistentQueue`s `EMPTY`** attribute.

```
(def pq (.-EMPTY PersistentQueue))
;; => #queue []
```

Using **`conj`** to add values to a queue adds items onto the rear:

```
(def pq (.-EMPTY PersistentQueue))
;; => #queue []

(conj (.-EMPTY PersistentQueue) 1 2 3)
;; => #queue [1 2 3]
```

A thing to bear in mind about queues is that the stack operations don't follow the usual stack semantics (pushing and popping from the same end), pops take values from the front position and pushes with **`conj`** append elements to the back.

```
(def pq (conj (.-EMPTY PersistentQueue) 1 2 3))
```

```
;; => #queue [1 2 3]

(peek pq)
;; => 1

(pop pq)
;; => #queue [2 3]

(conj pq 4)
;; => #queue [1 2 3 4]
```

Queues are not as frequently used as lists or vectors but is good to know that they are available in ClojureScript, they may come in handy some time.

3.8. Destructuring

Destructuring, as it name suggests, is a way of taking apart structured data such as collections and focusing on individual parts of them. ClojureScript offers a concise syntax for destructuring both indexed sequences and associative data structures that can be used in any place where bindings are declared.

Let’s see a an example of what destructuring is useful for, since it’ll help us understand the previous statements better. Imagine that you have a sequence but are only interested in the first and third item, you could get a reference to them easily with the **`nth`** function:

```
(let [v [0 1 2]
      fst (nth v 0)
      thrd (nth v 2)]
  [thrd fst])
;; => [2 0]
```

However, the previous code is overly verbose. Destructuring lets us extract values of indexed sequences more succintly using a vector in the left-hand side of a binding:

```
(let [[fst _ thrd] [0 1 2]]
  [thrd fst])
;; => [2 0]
```

In the above example, **`[fst _ thrd]`** is a destructuring form. It is represented as a vector and used for binding indexed values to the symbols **`fst`** and **`thrd`**, corresponding to the index **`0`** and **`2`** respectively. The _ symbol is used as a placeholder for indexes we are not interested in, in this case **`1`**.

Note that destructuring is not limited to the **`let`** binding form, it works in almost every place where we bind values to symbols such as in the **`for`** and **`doseq`** special forms or function arguments. We can write a function that takes a pair and swaps its positions very concisely using destructuring syntax in functionn arguments:

```
(defn swap-pair [[fst snd]]
  [snd fst])

(swap-pair [1 2])
;; => [2 1]

(swap-pair '(3 4))
;; => [4 3]
```

Positional destructuring with vectors is quite handy for taking indexed values out of sequences, but some times we don't want to discard the rest of the elements in the sequence when destructuring. Similarly to how **`&`** is used for accepting variadic function arguments, it can be used inside a vector destructuring form for grouping together the rest of a sequence:

```
(let [[fst snd & more] (range 10)]
  {:first fst
   :snd snd
   :rest more})
;; => {:first 0, :snd 1, :rest (2 3 4 5 6 7 8 9)}
```

Notice how the value in the **`0`** index got bound to **`fst`**, the value in the **`1`** index got bound to **`snd`** and the sequence of elements from **`2`** onwards got bound to the **`more`** symbol.

We may still be interested in a data structure as a whole even when we are destructuring it, this can be achieved with the **`:as`** keyword. If used inside a destructuring form, the original data structure is bound to the symbol following such keyword:

```
(let [[fst snd & more :as original] (range 10)]
  {:first fst
   :snd snd
   :rest more
   :original original})
```

```
;; => {:first 0, :snd 1, :rest (2 3 4 5 6 7 8 9), :original (0 1 2 3 4 5 6
 7 8 9)}
```

Not only indexed sequences can be destructured, associative data can also be destructured. Its destructuring binding form is represented as a map instead of a vector, where the keys are the symbols we want to bind values to and the values are the keys that we want to look up in the associative data structure. Let's see an example:

```
(let [{language :language} {:language "ClojureScript"}]
  language)
;; => "ClojureScript"
```

In the above example, we are extracting the value associated with the `:language` key and binding it to the `language` symbol. When looking up keys that are not present, the symbol will get bound to `nil`:

```
(let [{name :name} {:language "ClojureScript"}]
  name)
;; => nil
```

Associative destructuring lets us give default values to bindings, which will be used if the key isn't found in the data structure we are taking apart. A map following the `:or` keyword is used for default values, as the following examples show:

```
(let [{name :name :or {name "Anonymous"}} {:language "ClojureScript"}]
  name)
;; => "Anonymous"

(let [{name :name :or {name "Anonymous"}} {:name "Cirilla"}]
  name)
;; => "Cirilla"
```

Associative destructuring also supports binding the original data structure to a symbol placed after the `:as` keyword:

```
(let [{name :name :as person} {:name "Cirilla"}]
  [name person])
;; => ["Cirilla" {:name "Cirilla"}]
```

Not only keywords can be the keys of associative data structures. Numbers, strings, symbols and many other data structures can be used as keys, so we can destructure using those too. Note that we need to quote the symbols for preventing them from being resolved as a var lookup:

```
(let [{tenth 10} (range 100)]
  tenth)
;; => 11

(let [{name "name"} {"name" "Cirilla"}]
  name)
;; => "Cirilla"

(let [{lang 'language} {'language "ClojureScript"}]
  lang)
;; => "ClojureScript"
```

Since usually the values corresponding to keys are bound to their equivalent symbol representation (for example, when binding the value of **`:language`** to the **`language`**) and keys are usually keywords, strings or symbols ClojureScript offers shorthand syntax for these cases.

We'll show examples of all, starting with destructuring keywords using **`:keys`**:

```
(let [{:keys [name surname]} {:name "Cirilla" :surname "Fiona"}]
  [name surname])
;; => ["Cirilla" "Fiona"]
```

As you can see in the example, if we use the **`:keys`** keyword and associate it with a vector of symbols in a binding form, the values corresponding to the keywordized version of the symbols will be bound to them. The **`{:keys [name surname]}`** destructuring is equivalent to **`{name :name surname :surname}`**, only shorter.

The string and symbol shorthand syntax works exactly like **`:keys`**, but using the **`:strs`** and **`:syms`** keywords respectively:

```
(let [{:strs [name surname]} {"name" "Cirilla" "surname" "Fiona"}]
  [name surname])
;; => ["Cirilla" "Fiona"]

(let [{:syms [name surname]} {'name "Cirilla" 'surname "Fiona"}]
  [name surname])
;; => ["Cirilla" "Fiona"]
```

An interesting property of destructuring is that we can nest destructuring forms arbitrarily, which makes code that accesses nested data on a collection very easy to understand as it mimics the collection's structure:

```
(let [{[fst snd] :languages} {:languages ["ClojureScript" "Clojure"]}]
  [snd fst])
;; => ["Clojure" "ClojureScript"]
```

3.9. Namespaces

3.9.1. Defining a namespace

The *namespace* is ClojureScript's fundamental unit of code modularity. Namespaces are analogous to Java packages or Ruby and Python modules, and can be defined with the **ns** macro. Maybe if you have looked at a little bit of ClojureScript source you have seen something like this at begining of the file:

```
(ns myapp.core
  "Some docstring for the namespace.")

(def x "hello")
```

Namespaces are dynamic, meaning you can create one at any time. The convention however, is to have one namespace per file. Naturally, a namespace definition is usually at the beginning of the file, followed by an optional docstring.

Previously we have explained vars and symbols. Every var that you define will be associated with it's namespace. If you do not define a concrete namespace then the default one called "user" will be used:

```
(def x "hello")
;; => #'user/x
```

3.9.2. Loading other namespaces

Defining a namespace and the vars in it is really easy, but it's not very useful if we can't use them from other namespaces. For this purpose, the **ns** macro offers a simple way to load other namespaces.

Observe the following:

```
(ns myapp.main
  (:require myapp.core
            clojure.string))

(clojure.string/upper-case myapp.core/x)
;; => "HELLO"
```

As you can observe, we are using fully qualified names (namespace + var name) for access to vars and functions from different namespaces.

While this will let you access other namespaces, it's also repetitive and overly verbose. It will be especially uncomfortable if the name of a namespace is very large. To solve that, you can use the **`:as`** directive to create an additional (usually shorter) alias to the namespace. This is how it can be done:

```
(ns myapp.main
  (:require [myapp.core :as core]
            [clojure.string :as str]))

(str/upper-case core/x)
;; => "HELLO"
```

Additionaly, *ClojureScript* offers a simple way to refer to specific vars or functions from a concrete namespace using the **`:refer`** directive.

The **`:refer`** directive has two possible arguments: the **`:all`** keyword or a vector of symbols that will refer to vars in the namespace. With **`:all`**, we are indicating that we want to refer all public vars from the namespace, and with vector we can specify the specific subset of vars that we want. Effectively, it is as if those vars and functions are now part of your namespace, and you do not need to qualify them at all.

```
(ns myapp.main
  (:require [myapp.core :refer :all]
            [clojure.string :refer [upper-case]]))
(upper-case x)
;; => "HELLO"
```

And finally, you should know that everything located in the **`cljs.core`** namespace is automatically loaded and you should not require it explicitly. Sometimes you may want declare vars that will clash with some others defined in the **`cljs.core`** namespace. To do this, the **`ns`** macro offers another directive that allows you to exclude specific symbols and prevent them from being automatically loaded.

Observe the following:

```
(ns myapp.main
  (:refer-clojure :exclude [min]))

(defn min
  [x y]
  (if (> x y)
    y
    x))
```

The **`ns`** macro also has other directives for loading host classes (**`:import`**) and macros (**`:refer-macros`**), but these are explained in other sections.

3.10. Abstractions and Polymorphism

I'm sure that at more than one time you have found yourself in this situation: you have defined a great abstraction (using interfaces or something similar) for your "business logic" and you have found the need to deal with another module over which you have absolutely no control, and you probably were thinking of creating adapters, proxies, and other approaches that imply a great amount of additional complexity.

Some dynamic languages allow "monkey-patching"; languages where the classes are open and any method can be defined and redefined at any time. Also, it is well known that this technique is a very bad practice.

We can not trust languages that allow you to silently overwrite methods that you are using when you import third party libraries; you can not expect consistent behavior when this happens.

These symptoms are commonly called the "expression problem". see http://en.wikipedia.org/wiki/Expression_problem for more details

3.10.1. Protocols

The *ClojureScript* primitive for defining "interfaces" is called a Protocol. A protocol consists of a name and set of functions. All the functions have at least one argument corresponding to the **`this`** in javascript or **`self`** in Python.

Protocols provide a type-based polymorphism, and the dispatch is always done by the first argument (equivalent to JavaScript’s **`this`**, as previously mentioned).

A protocol looks like this:

```
(ns myapp.foobar)

(defprotocol IProtocolName
  "A docstring describing the protocol."
  (sample-method [this] "A doc string of the function associated with the
 protocol."))
```

> the "I" prefix is commonly used to designate the separation of protocols and types. In the clojure community there many different opinions about how the "I" prefix should be used. In our opinion, it is an acceptable solution to avoid name clashing and possible confusion.

From the user perspective, protocol functions are simply plain functions defined in the namespace where the protocol is defined. As you can intuit, this namespacing of protocols allows us to avoid any conflict between implemented protocols for the same type.

Extending existing types

One of the big strengths of protocols is the ability to extend existing and maybe third party types, and this operation can be done in different ways. The majority of time you will tend to use the **extend-protocol** or the **extend-type** macros.

This is an example of how the **extend-type** macro can be used:

```
(extend-type TypeA
  ProtocolA
  (function-from-protocol-a [this]
    ;; implementation here
    )

  ProtocolB
  (function-from-protocol-b-1 [this parameter1]
    ;; implementation here
    )
  (function-from-protocol-b-2 [this parameter1 parameter2]
    ;; implementation here
    ))
```

You can observe that with **extend-type** you are extending a single type with different protocols in a single expression. In comparison, **extend-protocol** does the inverse; given a protocol, it adds implementations for multiple types:

```
(extend-protocol ProtocolA
  TypeA
  (function-from-protocol-a [this]
    ;; implementation here
    )

  TypeB
  (function-from-protocol-a [this]
    ;; implementation here
    ))
```

There are other ways to extend a type with a protocol implementation, but they will be covered in another section of this book.

Participate in ClojureScript abstractions

ClojureScript it self is built up on abstractions defined as protocols. Almost all behavior in the *ClojureScript* language itself can be adapted to third party libraries. Let's go to see a real life example.

In previous sections we have explained the different kinds of built-in collections. For this example we will use the **Set**. See this snipped of code:

```
(def mynums #{1 2})

(filter mynums [1 2 4 5 1 3 4 5])
;; => (1 2 1)
```

What happened? In this case, the *set* type implements the *ClojureScript* internal **`IFn`** protocol that represents an abstraction for functions or anything callable. This way it can be used like a callable predicate in filter.

Ok, but what happens if we want use a regular expression as predicate function for filtering a collection of strings:

```
(filter #"^foo" ["haha" "foobar" "baz" "foobaz"])
;; TypeError: Cannot call undefined
```

The exception is raised because the RegExp type does not implements the **`IFn`** protocol so it cannot behave like a callable, but that can be easily fixed:

```
(extend-type js/RegExp
  IFn
  (-invoke
   ([this a]
     (re-find this a))))
```

Let's analyze this: we are extending the **`js/RegExp`** type so that it implements the **`invoke`** function in the **`IFn`** protocol. To invoke a regular expression **`a`** as if it were a function, call the **`re-find`** function with the object of the function and the pattern.

Now, you will be able use the regex instances as predicates in filter operation:

```
(filter #"^foo" ["haha" "foobar" "baz" "foobaz"])
;; => ("foobar" "foobaz")
```

Introspection using Protocols

ClojureScript comes with a useful function that allows runtime introspection: **`satisfies?`**. The purpose of this function is to determinate at runtime if some object (instance of some type) satisfies the concrete protocol.

So, with previous examples, if we check if a **`set`** instance satisfies a **IFn** protocol, it should return **`true`**:

```
(satisfies? IFn #{1})
;; => true
```

3.10.2. Multimethods

We have previously talked about protocols, which solve a very common use case of polymorphism: dispatch by type. But in some circumstances, the protocol approach it can be limiting. And here, **multimethods** come to the rescue.

These **multimethods** are not limited to type dispatch only; instead, they also offer dispatch by types of multiple arguments and by value. They also allow ad-hoc hierarchies to be defined. Also, like protocols, multimethods are an "Open System," so you or any third parties can extend a multimethod for new types.

The basic constructions of **multimethods** are the **`defmulti`** and **`defmethod`** forms. The **`defmulti`** form is used to create the multimethod with an initial dispatch function. This is a model of what it looks like:

```
(defmulti say-hello
  "A polymorphic function that return a greetings message
  depending on the language key with default lang as `:en`"
  (fn [param] (:locale param))
  :default :en)
```

The anonymous function defined within the **`defmulti`** form is a dispatch function. It will be called in every call to **`say-hello`** function and should return some kind of marker object that will be used for dispatch. In our example it returns the contents of the **`:locale`** key of the first argument.

And finally, you should add implementations. That is done with **`defmethod`** form:

```
(defmethod say-hello :en
  [person]
  (str "Hello " (:name person "Anonymous")))

(defmethod say-hello :es
  [person]
  (str "Hola " (:name person "Anónimo")))
```

So, if you execute that function over a hash map containing the **`:locale`** and optionally the **`:name`** key, the multimethod will first call the dispatch function to determine the dispatch value, then it will search for an implementation for that value. If an implementation is found, the dispatcher will execute it. Otherwise, the dispatch will search for a default implementation (if one is specified) and execute it.

```
(say-hello {:locale :es})
;; => "Hola Anónimo"

(say-hello {:locale :en :name "Ciri"})
;; => "Hello Ciri"
```

```
(say-hello {:locale :fr})
;; => "Hello Anonymous"
```

If the default implementation is not specified, an exception will be raised notifying you that some value does not have a implementation for that multimethod.

3.10.3. Hierarchies

Hierarchies are *ClojureScript*'s way to let you build whatever relations that your domain may require. The hierarchies are defined in term of relations between named objects, such as symbols, keywords or types.

The hierarchies can be defined globally or locally, depending on your needs. Like multimethods, hierarchies are not limited to a single namespace. You can extend a hierarchy from any namespace, not necessarily the one in which they are defined.

The global namespace is more limited, for good reasons. Keywords or symbols that are not namespaced can not be used in the global hierarchy. That behavior helps prevent unexpected situations when two or more third party libraries use the same symbol for different semantics.

Defining a hierarchy

The hierarchy relations should be established using the **`derive`** function:

```
(derive ::circle ::shape)
(derive ::box ::shape)
```

We have just defined a set of relationships between namespaced keywords. In this case the **`::circle`** is a child of **`::shape`**, and **`::box`** is also a child of **`::shape`**.

The **`::circle`** keyword syntax is a shorthand for **`:current.ns/circle`**. So if you are executing it in a REPL, **`::circle`** will be evaluated as **`:cljs.user/circle`**.

Hierarchies and introspection

ClojureScript comes with a little toolset of functions that allow runtime introspection of globally or locally defined hierarchies. This toolset consists of three functions: **`isa?`**, **`ancestors`**, and **`descendants`**.

Let's see an example of how it can be used with the hierarchy defined in previous example:

```
(ancestors ::box)
;; => #{:cljs.user/shape}

(descendants ::shape)
;; => #{:cljs.user/circle :cljs.user/box}

(isa? ::box ::shape)
;; => true

(isa? ::rect ::shape)
;; => false
```

Locally defined hierarchies

As we mentioned previously, in *ClojureScript* you also can define local hierarchies. This can be done with the **`make-hierarchy`** function. Here is an example of how you can replicate the previous example using a local hierarchy:

```
(def h (-> (make-hierarchy)
           (derive :box :shape)
           (derive :circle :shape)))
```

Now you can use the same introspection functions with that locally defined hierarchy:

```
(isa? h :box :shape)
;; => true

(isa? :box :shape)
;; => false
```

As you can observe, in local hierarchies we can use normal (not namespace qualified) keywords, and if we execute the **`isa?`** without passing the local hierarchy parameter, it returns **`false`** as expected.

Hierarchies in multimethods

One of the big advantages of hierarchies is that they works very well together with multimethods. This is because multimethods by default use the **`isa?`** function for the last step of dispatching.

Let’s see an example to clearly understand what that means. Firstly, define the multimethod with **`defmulti`** form:

```
(defmulti stringify-shape
  "A function that prints a human readable representation
  of a shape keyword."
  identity
  :hierarchy h)
```

With **`:hierarchy`** keyword parameter we indicate to the multimethod that hierarchy we want to use; if it is not specified, the global hierarchy will be used.

Secondly, we define an implementation for our multimethod using the **`defmethod`** form:

```
(defmethod stringify-shape :box
  [_]
  "A box shape")

(defmethod stringify-shape :shape
  [_]
  "A generic shape")

(defmethod stringify-shape :default
  [_]
  "Unexpected object")
```

Now, let’s see what happens if we execute that function with a box:

```
(stringify-shape :box)
;; => "A box shape"
```

Now everything works as expected; the multimethod executes the direct matching implementation for the given parameter. Next, let’s see what happens if we execute the same function but with the **`:circle`** keyword as the parameter, which does not have the direct matching dispatch value:

```
(stringify-shape :circle)
;; => "A generic shape"
```

The multimethod automatically resolves it using the provided hierarchy, and since **`:circle`** is a descendant of **`:shape`**, the **`:shape`** implementation is executed.

3.11. Data types

Until, now, we have used maps, sets, lists and vectors to represent our data. And in most cases, this is a really great approach. But sometimes we need to define our own types, and in this book we will call them **datatypes**.

A datatype provides the following:

- A unique host-backed type, either named or anonymous.
- The ability to implement protocols (inline).
- Explicitly declared structure using fields or closures.
- Map like behavior (via records, see below).

3.11.1. Deftype

The most low level construction in *ClojureScript* for creating your own types is the `deftype` macro. As a demonstration, we will define a type called `User`:

```
(deftype User [firstname lastname])
```

Once the type has been defined, we can create an instance of our `User`. In the following example, the `.` after `User` indicates that we are calling a constructor.

```
(def person (User. "Triss" "Merigold"))
```

And its fields can be accessed using the prefix-dot notation:

```
(.-firstname person)
;; => "Triss"
```

Types defined with `deftype` (and `defrecord`, which we will see later) creates a host-backed class-like object associated with the current namespace. But it has some peculiarities when we intend to use or import it from another namespace. The types in *ClojureScript* should be imported with the `:import` directive of the `ns` macro:

```
(ns myns.core
  (:import otherns.User))
```

```
(User. "Cirilla" "Fiona")
```

For convenience, *ClojureScript* also defines a constructor function caled `→User` that can be imported in the common way using the `:require` directive.

We personally do not like this type of function, and we prefer to define our own constructors, with more idiomatic names:

```
(defn make-user
  [firstname lastname]
  (User. firstname lastname))
```

And use it in our code instead of `→User`.

3.11.2. Defrecord

The record is a slightly higher level abstraction for defining types in *ClojureScript* and should be preferred way to do it.

As we know, *ClojureScript* tends to use plain data types such as maps, but in most cases we need a named type to represent the entities of our application. Here come the records.

A record is a datatype that implements the map protocol and therefore can be used like any other map. And since records are also proper types, they support type-based polymorphism through protocols.

In summary: with records, we have the best of both worlds, maps that can play in different abstractions.

Let start defining the `User` type but using records:

```
(defrecord User [firstname lastname])
```

It looks really similar to the `deftype` syntax; in fact, it uses `deftype` behind the scenes as a low level primitive for defining types.

Now, look at the difference with raw types for access to its fields:

```
(def person (User. "Yennefer" "of Vengerberg"))

(:firstname user)
```

```
;; => "Yennefer"

(get person :firstname)
;; => "Yennefer"
```

As we mentioned previously, records are maps and act like tham:

```
(map? person)
;; => true
```

And like maps, tham support extra fields that are not initially defined:

```
(def person2 (assoc person :age 92))

(:age person2)
;; => 92
```

As we can see, the **`assoc`** function works as expected and returns a new instance of the same type but with new key value pair. But take care with **`dissoc`**! Its behavior with records is slightly different than with maps; it will return a new record if the field being dissociated is an optional field, but it will return a plain map if you dissociate a mandatory field.

An other difference with maps is that records do not act like functions:

```
(def plain-person {:firstname "Yennefer", :lastname "of Vengerberg"})

(plain-person :firstname)
;; => "Yennefer"

(person :firstname)
;; => person.User does not implement IFn protocol.
```

For convenience, the **`defrecord`** macro, like **`deftype`**, exposes a **`→User`** function, as well as an additional **`map→User`** constructor function. We have the same opinion about that constructor as with **`deftype`** defined ones: we recommend defining your own instead of using the other ones. But as they exist, let’s see how they can be used:

```
(def cirilla (->User "Cirilla" "Fiona"))
(def yen (map->User {:firstname "Yennefer"
                     :lastname "of Vengerberg"}))
```

3.11.3. Implementing protocols

Both type definition primitives that we have seen so far allow inline implementations for protocols (explained in a previous section). Let's define one for example purposes:

```
(defprotocol IUser
  "A common abstraction for working with user types."
  (full-name [_] "Get the full name of the user."))
```

Now, you can define a type with inline implementation for an abstraction, in our case the **`IUser`**:

```
(defrecord User [firstname lastname]
  IUser
  (full-name [_]
    (str firstname " " lastname)))

;; Create an instance.
(def user (User. "Yennefer" "of Vengerberg"))

(full-name user)
;; => "Yennefer of Vengerberg"
```

3.11.4. Reify

The **`reify`** macro lets you create an anonymous type that implements protocols. In contrast to **`deftype`** and **`defrecord`**, it does not have accessible fields.

This is how we can emulate an instance of the user type that plays well with the **`IUser`** abstraction:

```
(defn user
  [firstname lastname]
  (reify
    IUser
    (full-name [_]
      (str firstname " " lastname))))

(def yen (user "Yennefer" "of Vengerberg"))
(full-name user)
;; => "Yennefer of Vengerberg"
```

The real purpose of **`reify`** is to create an anonymous type that may implement protocols, but you don't want the type itself.

3.12. Host interoperability

ClojureScript, in the same way as it brother Clojure, is designed to be a "Guest" language. This means that the design of the language works well on top of an existing ecosystem such as JavaScript for *ClojureScript* and the JVM for *Clojure*.

3.12.1. The types.

ClojureScript unlike what you might expect, tries to take advantage of every type that the platform provides. This is a (perhaps incomplete) list of things that *ClojureScript* inherits and reuses from the underlying platform:

- *ClojureScript* strings are javascript **Strings**.
- *ClojureScript* numbers are javascript **Numbers**.
- *ClojureScript* **`nil`** is a javascript **null**.
- *ClojureScript* regular expressions are javascript **`RegExp`** instances.
- *ClojureScript* is not interpreted; it is always compiled down to JavaScript.
- *ClojureScript* allows easy call to platform APIs with the same semantics.
- *ClojureScript* data types internally compile to objects in JavaScript.

On top of it, *ClojureScript* builds its own abstractions and types that do not exist in the platform, such as Vectors, Maps, Sets, and others that are explained in previous sections of this chapter.

3.12.2. Interacting with platform types

ClojureScript comes with a little set of special forms that allows it interact with platform types such as calling object methods, creating new instances, and accessing object properties.

Access to the platform

ClojureScript has a special syntax for access to the entire platform environment through the **`js/`** special namespace. This is an example of an expression to execute JavaScript's built-in **`parseInt`** function:

```
(js/parseInt "222")
;; => 222
```

Creating new instances

ClojureScript has two ways to create instances:

Using the new special form

```
(new js/RegExp "^foo$")
```

Using the **.** special form

```
(js/RegExp. "^foo$")
```

The last one is the recommended way to create instances. We are not aware of real differences between the two forms, but in the ClojureScript community the last one is used most often.

Invoke instance methods

To invoke methods of some object instance, as opposed to how it is done in JavaScript (e.g., **`obj.method()`**, the method name comes first like any other standard function in Lisp languages but with a little variation: the function name starts with special form **`.`**.

Let's see how we can call the **`.test()`** method of a regexp instance:

```
(def re (js/RegExp "^Clojure"))

(.test re "ClojureScript")
;; => true
```

Access to object properties

Access to the object properties is really very similar to calling a method. The difference is that instead of using the **`.`** you use **`.-`**. Let's see an example:

```
(.-multiline re)
;; => false
(.-PI js/Math)
;; => 3.141592653589793
```

JavaScript objects

ClojureScript has different ways to create plain JavaScript objects, each one has its own purpose. The basic one is the **`js-obj`** function. It accepts a variable number of pairs of keys and values and returns a JavaScript object:

```
(js-obj "country" "FR")
;; => #js {:country "FR"}
```

The return value can be passed to some kind of third party library that accepts a plain JavaScript object, but you can observe the real representation of the return value of this function. It is really another other form for doing the same thing.

Using the reader macro **`#js`** consists of prepending it to a ClojureScript map or vector, and the result will be transformed to plain JavaScript:

```
(def myobj #js {:country "FR"})
```

The translation of that to plain javascript is similar to this:

```
var myobj = {country: "FR"};
```

As explained in previous section, you also can access to the plain object properties using the **`.-`** syntax:

```
(.-country myobj)
;; => "FR"
```

And as JavaScript objects are mutable, you can set a new value for some property using the **`set!`** function:

```
(set! (.-country myobj) "KR")
```

Conversions

The inconvenience of the previously explained forms is that they does not make recursive transformations, so if you have nested objects, the nested objects will not be converted. To solve that use cases *ClojureScript* comes with the **`clj→js`** and **`js→clj`** functions that transform clojure collection types into JavaScript and back:

```
(clj->js {:foo {:bar "baz"}})
;; => #js {:foo #js {:bar "baz"}}
```

In case of arrays, there is a specialized function **`into-array`** that behaves as expected:

```
(into-array ["France" "Korea" "Peru"])
;; => #js ["France" "Korea" "Peru"]
```

Arrays

In previous example we have seen how we can create an array from an existing *ClojureScript* collection. But there is another function for creating arrays: **`make-array`**.

Creating a preallocated array with length 10

```
(def a (make-array 10))
;; => #js [nil nil nil nil nil nil nil nil nil nil]
```

In *ClojureScript* arrays also play well with sequence abstractions so you can iterate over them or simply get the number of elements with the **`count`** function:

```
(count a)
;; => 10
```

As arrays on the JavaScript platform are a mutable collection type, you can access a concrete index and set the value at that position:

```
(aset a 0 2)
;; => 2
a
;; => #js [2 nil nil nil nil nil nil nil nil nil]
```

Or access in a indexed way to get its values:

```
(aget a 0)
;; => 2
```

In JavaScript, objects are also arrays, so you can use the same functions for interacting with plain objects:

```
(def b #js {:hour 16})
;; => #js {:hour 16}

(aget b "hour")
;; => 16

(aset b "minute" 22)
;; => 22

b
;; => #js {:hour 16, :minute 22}
```

3.13. State management

TBD

3.14. Truthiness

This is the aspect where each language has its own semantics. The majority of languages consider empty collections, the integer 0, and other things like this to be false. In *ClojureScript*, unlike in other languages, only two values are considered as false: **`nil`** and **`false`**, Everything else is treated as **`true`**.

So, thanks to this, sets can be also be considered as predicates. If a set returns a value, it exists, and if it returns **`nil`** the value does not exist:

```
(def s #{1 2})

(s 1)
;; => 1

(s 3)
;; => nil
```

3.15. Transducers

TBD

3.16. Transients

TBD

3.17. Metadata

TBD

3.18. A little overview of macros

TBD

Chapter 4. Tooling & Compiler

This chapter will cover a little introduction to existing tooling for making things easy when developing using ClojureScript. It will cover:

- Using the repl
- Leiningen and cljsbuild
- Google Closure Library
- Modules
- Unit testing
- Library development
- Browser based development
- Server based development

Unlike the previous chapter, this chapter intends to tell different stories each independent of each other.

4.1. Getting Started with the ClojureScript compiler

At this point, you are surely very bored with the constant theoretical explanations about the language itself and will want to write and execute some code. The goal of this section is to provide a little practical introduction to the *ClojureScript* compiler.

The *ClojureScript* compiler takes the source code that has been split over numerous directories and namespaces and compiles it down to JavaScript. Today, JavaScript has great number of different environments where it can be executed - each with its own pecularities.

This chapter intends to explain how to use *ClojureScript* without any additional tooling. This will help you understand how the compiler works and how you can use it when other tooling is not available (such as leiningen+cljsbuld or boot).

4.1.1. Execution environments

What is an execution environment? An execution environment is an engine where JavaScript can be executed. For example, the most popular execution environment is a browser (Chrome, Firefox, …) followed by the second most popular - nodejs/iojs.

There are others, such as Rhino (jdk6+), Nashorn (jdk8), QtQuick (QT),… but none of them have significant differences from the first two. So, *ClojureScript* at the moment may compile code to run in the browser or in nodejs/iojs like environments out of the box.

4.1.2. Download the compiler

The *ClojureScript* compiler is implemented in java, and to use it, you should have jdk8 installed. *ClojureScript* itself only requires jdk7, but the standalone compiler that we going to use in this chapter requires jdk8 which can be found at http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

You can download it using **`wget`**:

```
wget https://github.com/clojure/clojurescript/releases/download/r3211/
cljs.jar
```

The *ClojureScript* compiler is packaged in a standalone executable jar file, so this is the only file and JDK8 that you need to compile the *ClojureScript* source code to JavaScript.

4.1.3. Compile for nodejs/iojs

Let start with a practical example compiling code that will target **nodejs/iojs**. For this example you should have nodejs or iojs (recommended) installed.

There are different ways to install iojs, but the recommended way is using nvm (node version manager). You can read the instructions to install and use nvm at: home page[1].

You can test if **iojs** is installed in your system with this command:

```
$ iojs --version
v1.7.1
```

Create the example application

For the first step of our practical example, we will create our application directory structure and populate it with example code.

So start creating the directory tree structure for our "hello world" application:

[1] https://github.com/creationix/nvm

```
mkdir -p src/myapp
touch src/myapp/core.cljs
```

Second, write the example code into the previously created **`src/myapp/core.cljs`** file:

```
(ns myapp.core
  (:require [cljs.nodejs :as nodejs]))

(nodejs/enable-util-print!)

(defn -main [& args]
  (println "Hello world!"))

(set! *main-cli-fn* -main)
```

It is very important that the declared namespace in the file exactly matches the directory structure. This is the way *ClojureScript* structures its source code.

Compile the example application

In order to compile that source code, we need a simple build script that instructs the *ClojureScript* compiler with the source directory and the output file. *ClojureScript* has a lot of other options but at this moment we can ignore that.

Lets create the *build.clj* file with the following content:

```
(require 'cljs.closure)

(cljs.closure/build "src"
 {:output-to "main.js"
  :main 'myapp.core
  :target :nodejs})
```

This is a brief explanation of the compiler options used in this example:

- The **`:output-to`** parameter indicates to the compiler the destination of the compiled code, in this case to the "main.js" file.
- The **`:main`** property indicates to the compiler the namespace that will act as the entry point of your application when it’s executed.

- The **`:target`** property indicates the platform where you want execute the compiled code. In this case we are going to use **iojs** (formerly nodejs). If you omit this parameter, the source will be compiled to run in the browser environment.

To run the compilation, just execute the following command:

```
java -cp cljs.jar:src clojure.main build.clj
```

And when it finishes, execute the compiled file using **iojs**:

```
$ iojs main.js
Hello world!
```

4.1.4. Compile for the Browser

In this section we are going to create a similar "hello world" example application from the previous section to run in the browser environent. The minimal requirement for it is just a browser that can execute JavaScript.

The process is almost the same, the directory structure is the same. The only things that changes is the entry point of the application and the build script.

Start writing new content to the **`src/myapp/core.cljs`** file:

```
(ns myapp.core)

(enable-console-print!)

(println "Hello world!")
```

In the browser environment we do not need a specific entry point for the application, so the entry point is the entire namespace.

Compile the example application

In order to compile the source code to run properly in a browser, overwrite the *build.clj* file with the following content:

```
(require 'cljs.closure)

(cljs.closure/build "src"
 {:output-to "main.js"
  :output-dir "out/"
  :source-map "main.js.map"
  :main 'myapp.core
  :optimizations :none})
```

This is a brief explanation of the compiler options we’re using:

- The **`:output-to`** parameter indicates to the compiler the destination of the compiled code, in this case to the "main.js" file.
- The **`:main`** property indicates to the compiler the namespace that will act as the entry point of your application when it’s executed.
- **`:source-map`** indicates the destination of the source map. (The source map connects the ClojureScript source to the generated JavaScript so that error messages can point you back to the original source.)
- **`:output-dir`** indicates the destination directory for all files sources used in a compilation. It is just for making source maps work properly with the rest of code, not only your source.
- **`:optimizations`** indicates the compilation optimization. There are different values for this option, but that will be covered in following sections in more detail.

To run the compilation, just execute the following command:

```
java -cp cljs.jar:src clojure.main build.clj
```

This process can take some time, so do not worry, wait a little bit. The JVM bootstrap with Clojure compiler is slightly slow. In the following sections we will explain how to start a watch process to avoid constantly starting and stopping this slow process.

While waiting for the compilation, let’s create a dummy HTML file to make it easy to execute our example app in the browser. Create the *index.html* file with the following content; it goes in the main *myapp* directory.

```
<!DOCTYPE html>
<html>
  <header>
    <meta charset="utf-8" />
    <title>Hello World from ClojureScript</title>
  </header>
  <body>
```

```
    <script src="main.js"></script>
  </body>
</html>
```

Now, when the compilation finishes and you have the basic HTML file you can just open it with your favorite browser and take a look in the development tools console. There should appear the "hello world" message.

4.1.5. Watch process

Surely, you have already experienced the slow startup of the *ClojureScript* compiler. To solve this, the *ClojureScript* standalone compiler also comes with tools to start a process that watches the changes in some directory and perform an incremental compilation.

Start creating another build script, but in this case name it *watch.clj*:

```
(require 'cljs.closure)

(cljs.closure/watch "src"
 {:output-to "main.js"
  :output-dir "out/"
  :source-map "main.js.map"
  :main 'myapp.core
  :optimizations :none})
```

Now, execute that script like any other that you have executed in previous sections:

```
$ java -cp cljs.jar:src clojure.main watch.clj
Building ...
Reading analysis cache for jar:file:/home/niwi/cljsbook/playground/
cljs.jar!/cljs/core.cljs
Compiling out/cljs/core.cljs
Using cached cljs.core out/cljs/core.cljs
... done. Elapsed 0.8354759 seconds
Watching paths: /home/niwi/cljsbook/playground/src

Change detected, recompiling ...
Compiling src/myapp/core.cljs
Compiling out/cljs/core.cljs
Using cached cljs.core out/cljs/core.cljs
... done. Elapsed 0.191963443 seconds
```

You can observe that in the second compilation, the time is drastically reduced. Another advantage of this method is that it is a gives a little bit more output.

4.1.6. Optimization levels

The *ClojureScript* compiler has different level of optimizations. Behind the scenes, those compilation levels are coming from Google Closure Compiler.

A very simplified overview of the compilation process is:

1. The reader reads the code and makes some analysis. This process can raise some warnings during its phase.
2. Then, the *ClojureScript* compiler emits JavaScript code. The result of that is one JavaScript file for each cljs file.
3. The generated files passes through the Closure Compiler that depending on the optimization level, and other options (sourcemaps, output dir output to, …) generates the final output.

The final output depends strictly on the optimization level.

none

Implies that closure compiler just writes the files as is, without any additional optimization applied to the source code. This optimization level is mandatory if you are targeting **nodejs** or **iojs** and is appropiate in development mode when your code targets the browser.

whitespace

This optimization level consists of concatenating the compiled files in an appropriate order, removing line breaks and other whitespace and generating the output as one large file.

It also has some compilation speed penalty, resulting in slower compilations. In any case, it is not terribly slow and is completely usable in small/medium applications.

simple

The simple compilation level implies (includes) all transformations from whitespace optimization and additionally performs optimizations within expressions and functions, including renaming local variables and function parameters to shorter names.

Compilation with the **`:simple`** optimization always preserves the functionality of syntactically valid JavaScript, so it does not interfere with the interaction between the compiled *ClojureScript* and other JavaScript.

advanced

TBD

4.2. Working with the REPL

4.2.1. Introduction

Although you can create a source file and compile it every time you want to try something out in ClojureScript, it’s easier to use the REPL. REPL stands for:

- Read - get input from the keyboard
- Evaluate the input
- Print the result
- Loop back for more input

In other words, the REPL lets you try out ClojureScript concepts and get immediate feedback.

4.2.2. Node REPL

TBD

4.2.3. Nashorn REPL

TBD

4.2.4. Browser REPL

Let start creating a file named **`brepl.clj`** with the following content, adapted from the [ClojureScript Quick Start Wiki Page](https://github.com/clojure/clojurescript/wiki/Quick-Start).

```
(require 'cljs.repl)
(require 'cljs.closure)
(require 'cljs.repl.browser)
```

```
(cljs.closure/build "src"
  {:main 'myapp.core
   :output-to "main.js"
   :verbose true})

(cljs.repl/repl (cljs.repl.browser/repl-env)
  :watch "src"
  :output-dir "out/")
```

This script builds the source, just as we did earlier, and then starts the REPL. It then runs the REPL. watching the source directory for any changes. The REPL starts a mini web server to display your web page. You must change your ClojureScript code to connect your script to the REPL. Change the source code in *src/myapp/core.cljs* to read:

```
(ns myapp.core
 (:require [clojure.browser.repl :as repl]))

(defonce conn
  (repl/connect "http://localhost:9000/repl"))

(enable-console-print!)

(println "Hello, world!")
```

The **`:require`** has changed to use code from the **`clojure.browser.repl`** library, and then defines a connection to the REPL, which will be running on port 9000 of ClojureScript's mini-server.

There's one more step to take. On Linux and Mac OSX, the REPL doesn't automatically let you use the arrow keys to move back and forth within a line or up and down to see previously entered input. Instead, you need to use the **`rlwrap`** program to give you that capability. (Its name comes from the fact that it wraps the "readline" utility around other programs.)

Well, that was a lot of setup! But trust us, it's all worth it when you see it in action. Here it is, with a ClojureScript expression typed in as an example:

```
$ rlwrap java -cp cljs.jar:src clojure.main brepl.clj
Compiling client js ...
Waiting for browser to connect ...
Watch compilation log available at: out/watch.log
To quit, type: :cljs/quit
```

```
cljs.user=> (+ 14 28)
42
cljs.user=>
```

Since you are watching the source code, if you make changes, your program will recompile, and the compiler output will go to the *watch.log* file in the *out* directory. If you are on Mac OSX or Linux, you can open up a new terminal window and use this command to see the contents of the file continuously updated:

```
tail -f out/watch.log
```

4.3. Build and Dependency management tools

4.3.1. Getting started with Leiningen.

TBD

4.3.2. Getting started with boot.

TBD

4.4. The Closure Library

TBD

4.5. Browser based development

TBD

4.5.1. Using third party JavaScript libraries

TBD

4.5.2. Modularizing your code

TBD

4.6. Developing a library

TBD

4.7. Unit testing

TBD

Chapter 5. Mixed Bag

This chapter will cover miscellaneous topics that are not classified in the previous ones. This is a "catchall" section and will touch a bunch of heterogeneus topics like:

- Async primitives using the *core.async* library.
- Working with promises.
- Error handling using the *cats* library.
- Pattern matching with the *core.match* library.
- Web development using the Om library.
- Share code between clojure and clojurescript.

5.1. Async primitives using core.async.

TBD

5.2. Working with promises.

TBD

5.3. Error handling using monads and Cats.

TBD

5.4. Pattern matching using core.match.

TBD

5.5. Web development with Om and React.

TBD

5.6. Writing libraries that shares code between Clojure and ClojureScript.

TBD