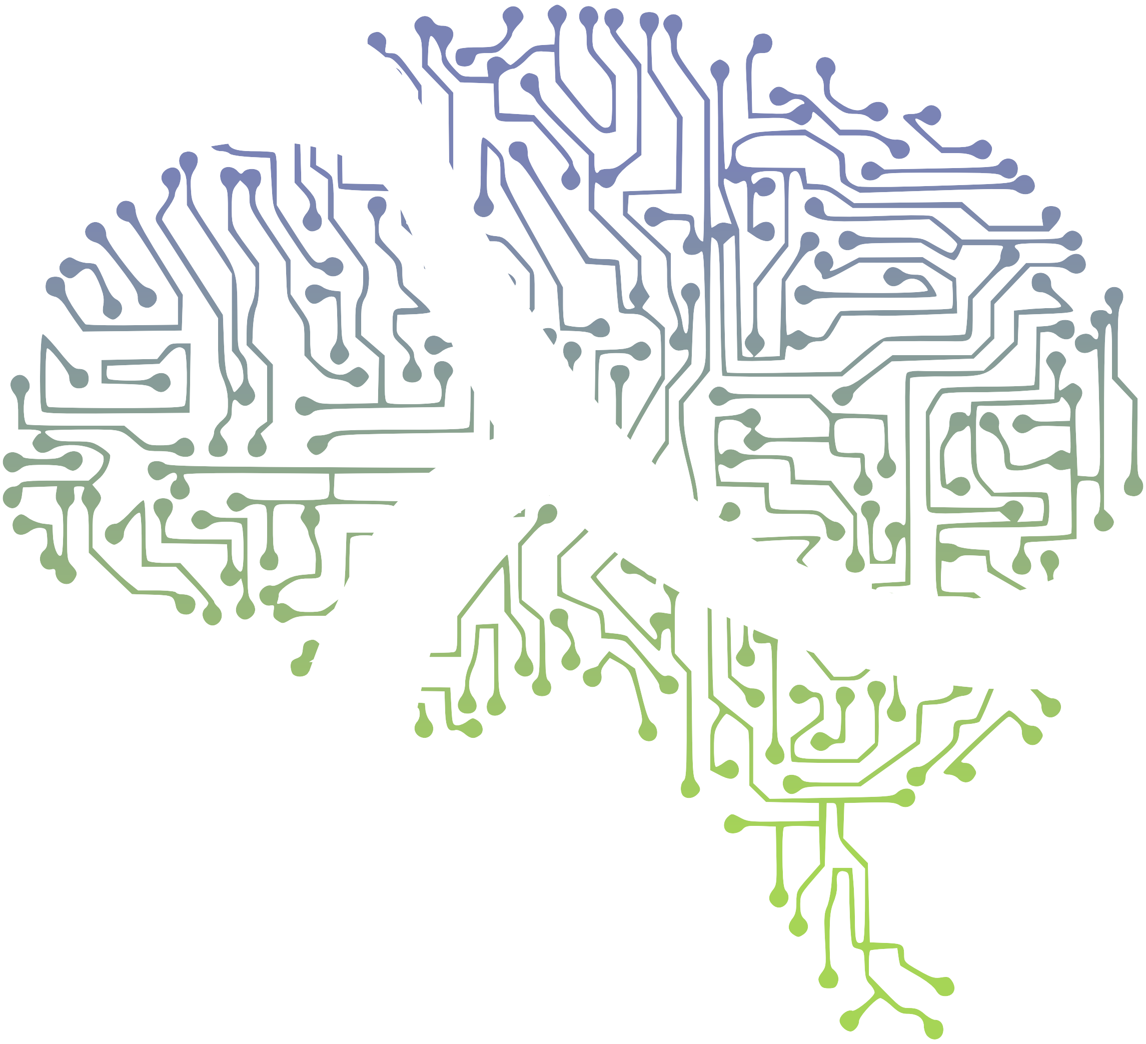


# ClojureScript Unraveled

(& "Andrey Antukh"
"Alejandro Gómez")

# ClojureScript Unraveled

Andrey Antukh <niwi@niwi.nz>

Alejandro Gómez <alejandro@dialelo.com>

Revision 3

2015-08-02

# Table of Contents

# Chapter 1. About this book

This book covers the ClojureScript programming language, serves as a detailed guide of its tooling for development, and presents a series of articles about topics that are applicable to 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 underpinnings of ClojureScript that may not be familiar to everyone.

Since the ClojureScript documentation is good but sparse, 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 as a sort of cookbook with practical programming recipes.

You’ll get the most out of this book if you:

- are curious about ClojureScript or functional programming and have some programming experience;
- write JavaScript or any other language that compiles to it and want to know what ClojureScript has to offer;
- 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 any of the above groups. We encourage you to give this book a try and to give us feedback on how we can make it more accessible. 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

> Why are we doing this? Because Clojure *rocks*, and JavaScript *reaches*.
>
> — Rich Hickey

*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 Nashorn.

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. Its semantics differ significantly from those of JavaScript.

Another big difference (and in our opinion an 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 easily adapted to the nuances of its execution environment. This has the benefit that Clojure (and hence ClojureScript) has access to all the existing libraries written for the host language.

Before we jump in, let us summarize some of the core ideas that ClojureScript brings to the table. Don't worry if you don't understand all of them right now, they'll become clear throughout the book.

- ClojureScript enforces the functional programming paradigm with its design decisions and idioms. Although being strongly opinionated about functional programming it's a pragmatic language rather than pursuing theoretical purity.
- Encourages programming with immutable data, offering highly performant and state of the art immutable collection implementations.
- It makes a clear distinction of identity and its state, with explicit constructs for managing change as a series of immutable values over time.
- It has type-based and value-based polymorphism, elegantly solving the expression problem.
- It is a Lisp dialect so programs are written in the programming language's own data structures, a property known as *homoiconicity* that makes metaprogramming (programs that write programs) as simple as it can be.

These ideas together have a great influence in the way you design and implement software, even if you are not using Clojure. Functional programming, decoupling of data (which is immutable) from the operations to transform it, explicit idioms for managing change over time and polymorphic constructs for programming to abstractions greatly simplify the systems we write.

> We can make the same exact software we are making today with dramatically simpler stuff — dramatically simpler languages, tools, techniques, approaches.
>
> — Rich Hickey

We hope you enjoy the book and ClojureScript brings the same joy and inspiration that has brought to us.

# Chapter 3. Language (the basics)

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 about clojurescript and 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, ClojureScript being one of them. It is 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. In the example below, we apply the addition function to three arguments. Note that unlike in other languages, **+** is not an operator but a function. Lisp has no operators; it only has functions.

```
(+ 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
```

To distinguish function calls from lists of data items, 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 to extend 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 data types 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, regexes (regular expressions), vars, atoms, and volatiles.

*ClojureScript* embraces the host language, and where possible, uses the host's provided types. For example: numbers and strings are used as is and behave in the 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 the following ways:

```
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; functions are discussed in a later section.

```
(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, which can be functions, variables, etc.

Symbols start with a non numeric character and can contain alphanumeric characters as well as *, +, !, -, _, ', and ? such as :

```
sample-symbol
```

```
othersymbol
f1
my-special-swap!
```

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:

```
"An 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 newline 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 collections. The main difference between *ClojureScript* collections and collections in other languages is that they are persistent and immutable.

Before moving on to 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 something callable, and the rest of the list elements will be function arguments. 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 can 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 the first argument, 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 peculiarity 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 its 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 sections 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 allow you to store key/value pairs. In other languages, this type of structure is commonly known as a hash-map or dict (dictionary). 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 they 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. Like maps, they use curly braces for their literal syntax, with the difference being that they use a `#` as the leading character. You can also use the `set` function to convert a collection to a set:

```
#{1 2 3 :foo :bar}
;; => #{1 :bar 3 :foo 2}
(set [1 2 1 3 1 4 1 5])
;; => #{1 2 3 4 5}
```

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

## 3.3. Vars

*ClojureScript* is a mostly functional language that focuses 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 the **`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 another section.

If you want to 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
```

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)
```

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

```
;; => 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 an operator; everything is a function. The immediate advantage is that the prefix notation allows an arbitrary number of arguments per "operator". It also completely eliminates the problem of operator precedence.

### 3.4.2. Defining your own functions

You can define an unnamed (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 the 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; 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:

```clojure
(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. Functions with multiple arities

*ClojureScript* also comes with the ability to define functions with an 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 explain it better:

```clojure
(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 the wrong number of arguments, the compiler will emit an error message.

```clojure
(myinc 1)
;; => 2

(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 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 the **fn** special form.

```
(def average #(/ (+ %1 %2) 2))

(average 3 4)
;; => 3.5
```

The preceding definition is shorthand for:

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

```
(def average-longer (fn [a b] (/ (+ a b) 2)))

(average-longer 7 8)
;; => 7.5
```

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.

If a function only accepts one argument, you can omit the number after the **%** symbol, e.g., a function that squares a number: **#(* %1 %1))** can be written **#(* % %))**.

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 to flow control than languages like JavaScript, C, etc.

### 3.5.1. Branching with **if**

Let's start with a basic one: **if**. In *ClojureScript*, the **if** is an expression and not a statement, and it has three parameters: 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 limiting because it does not have the "else if" part to add more than one condition. The **cond** macro 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 result of evaluating **(keyword code)**.

### 3.5.3. Branching with **case**

The **case** branching expression has a similar use as our previous example with **condp**. The main differences are 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 static.

Here is the previous example rewritten to use **case**:

```
(defn translate-lang-code
  [code]
  (case code
    "es" "Spanish"
    "en" "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 have the concept of variables as in ALGOL-like languages, but it does have locals. Locals, as per usual, are immutable, and if you try to mutate them, the compiler will throw an error.

Locals are defined with the **let** expression. The expression starts with a vector as the first parameter followed by an 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 the 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.

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 rerun with **`recur`**, incrementing the binding value with the **`inc`** function. We do this once more until the condition is met and, since there aren't any 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 with 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)
```

The first parameter for **`map`** can be *any* function that takes one argument and returns a value. For example, if you had a graphing application and you wanted to graph the equation **`y = 3x + 5`** for a set of *x* values, you could get the *y* values like this:

```
(defn y-value [x] (+ (* 3 x) 5))

(map y-value [1 2 3 4 5])
;; => (8 11 14 17 20)
```

If your function is short, you can use an anonymous function instead, either the normal or short syntax:

```
(map (fn [x] (+ (* 3 x) 5)) [1 2 3 4 5])
;; => (8 11 14 17 20)
(map #(+ (* 3 %) 5) [1 2 3 4 5])
;; => (8 11 14 17 20)
```

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)
```

Again, you can use any function that returns **`true`** or **`false`** as the first argument to **`filter`**. Here is an example that keeps only words less than five characters long. (The **`count`** function returns the length of its argument.)

```
(filter (fn [word] (< (count word) 5))
 ["ant" "baboon" "crab" "duck" "echidna" "fox"])
;; => ("ant" "crab" "duck" "fox")
```

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
```

Yet again, you can provide your own function as the first argument to **`reduce`**, but your function must have *two* parameters. The first one is the "accumulated value" and the second parameter is the collection item being processed. The function returns a value that becomes the accumulator for the next item in the list. For example, here is how you would find the sum of squares of a set of numbers (this is an important calculation in statistics). Using a separate function:

```
(defn sum-squares [accumulator item]
  (+ accumulator (* item item)))

(reduce sum-squares 0 [3 4 5])
;; => 50
```

…and with an anonymous function:

```
(reduce (fn [acc item] (+ acc (* item item))) 0 [3 4 5])
;; => 50
```

Here is a **`reduce`** that finds the total number of characters in a set of words:

```
(reduce (fn [acc word] (+ acc (count word))) 0
 ["ant" "bee" "crab" "duck"])
;; => 14
```

We have not used the short syntax here because, although it requires less typing, it can be less readable, and when you are starting with a new language, it's important to be able to read what you wrote! If you are comfortable with the short syntax, feel free to use it.

Remember to choose your starting value for the accumulator carefully. If you wanted to use **`reduce`** to find the product of a series of numbers, you would have to start with one rather than zero, otherwise all the numbers would be multiplied by zero!

```
;; wrong starting value
```

```
(reduce * 0 [3 4 5])
;; => 0

;; correct starting accumulator
(reduce * 1 [3 4 5])
;; => 60
```

## `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 an 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 x)])
;; => ([1 1] [2 4] [3 9])
```

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 squared.

**`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 suggests, is a data structure that cannot 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`** (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

Since **seq** returns **nil** when the collection is empty, and **nil** evaluates to false in boolean context, you can check to see if a collection is empty by using the **seq** function. The technical term for this is 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
```

Though **nil** is neither a seqable nor a sequence, it is supported by 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 section. This makes them highly generic because 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)
```

Note: When you use the **map** function on a map collection, your higher-order function will receive a two-item vector containing a key and value from the map. The following example uses destructuring to access the key and value.

```
(map (fn [[key value]] (* value value)) {:ten 10 :seven 7 :four 4})
;; => (100 49 16)
```

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 a 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. This operation also works on strings, even though, as you have seen, they are not collections, sequences, or seqable.

```
(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 type of 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 as 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.

Consider the **`range`** function, which generates a range of integers:

```
(range 5)
;; => (0 1 2 3 4)
(range 1 10)
;; => (1 2 3 4 5 6 7 8 9)
(range 10 100 15)
;; (10 25 40 55 70 85)
```

If you just say **`(range)`**, you will get an infinite sequence of all the integers. Do **not** try this in the REPL, unless you are prepared to wait for a very, very long time, because the REPL wants to fully evaluate the expression.

Here is a contrived example. Let's say you are writing a graphing program and you are graphing the equation *y*= 2_x_$^{2}$ + 5, and you want only those values of *x* for which the *y* value is less than 100. You can generate all the numbers 0 through 100, which will certainly be enough, and then **`take-while`** the condition holds:

```
(take-while (fn [x] (< (+ (* 2 x x) 5) 100)) (range 0 100))
;; => (0 1 2 3 4 5 6)
```

## 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 it 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 items to 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, 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 **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 is 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 also be used as stacks 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, particularly 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 they 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"}
```

In the preceding example, it just so happens that the order was preserved, but if you have many keys, you will see that the order is not preserved.

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**. **assoc** will also update the value of an existing key. Let's explore these functions:

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

Maps are also functions of their keys, returning the values related to the given keys. 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 items to compare and return -1 (if the first item is less than the second), 0 (if they are equal), or 1 (if the first item is greater than the second).

```
(defn reverse-compare [a b] (compare b a))

(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 cannot contain duplicate values. If you accidentally 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 they 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 just like 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}
```

Sets act as read-only associative data that associates the values it contains to themselves. Since every value except **`nil`** and **`false`** is truthy 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"
```

```
(vowels \z)
;; => nil

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

Sets have a sorted counterpart like maps do that are 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 ordered-set)
;; => ([0] [1] [2])
```

## Queues

ClojureScript also provides a persistent and immutable queue. Queues are not used as pervasively as other collection types. They can be created using the **`#queue []`** literal syntax, but there are no convenient constructor functions for them.

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

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

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

(conj pq 4 5)
;; => #queue [1 2 3 4 5]
```

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). **`pop`** takes values from the front position, and **`conj`** pushes (appends) elements to the back.

```
(def pq #queue [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 it is good to know that they are available in ClojureScript, as they may occasionally come in handy.

## 3.8. Destructuring

Destructuring, as its 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 any place where bindings are declared.

Let's see an example of what destructuring is useful for that will 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 on 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 in function arguments. We can write a function that takes a pair and swaps its positions very concisely using destructuring syntax in function 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 sometimes 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, the ampersand 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 that 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 can indexed sequences be destructured, but 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" :age 49}]
  [name person])
;; => ["Cirilla" {:name "Cirilla" :age 49}]
```

Keywords aren't the only things that 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 to prevent them from being resolved as a var lookup:

```
(let [{one 1} {0 "zero" 1 "one"}]
  one)
;; => "one"

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

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

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

We'll show examples of all of these, 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. If you have ever looked at a little bit of ClojureScript source, you may have noticed something like this at the beginning 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. However, the convention 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 its namespace. If you do not define a concrete namespace, then the default one called "cljs.user" will be used:

```
(def x "hello")
;; => #'cljs.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 symbols 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 long. 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"
```

Additionally, *ClojureScript* offers a simple way to refer to specific vars or functions from a concrete namespace using the **`:refer`** directive, followed by a sequence of symbols that will refer to vars in the namespace. 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 [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 to 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.

### 3.9.3. Namespaces and File Names

When you have a namespace like **`myapp.core`**, the code must be in a file named *core.cljs* inside the *myapp* directory. So, the preceding examples with namespaces **`myapp.core`** and **`myapp.main`** would be found in project with a file structure like this:

```
myapp
└── src
    └── myapp
        ├── core.cljs
        └── main.cljs
```

## 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 cannot 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.testproto)
```

```
(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 are 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 different protocols implemented for the same type.

Here is an example. Let’s create a protocol called `IInvertible` for data that can be "inverted". It will have a single method named `invert`.

```
(ns proto.testproto)

(defprotocol IInvertible
  "This is a protocol for data types that are 'invertible'"
    (invert [this] "Invert the given item."))
```

## 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. Here is code that will extend the **`number`**, **`string`**, **`List`**, and **`PersistentVector`** types to be "invertible". For numbers, we define the inverse to be the reciprocal of the number (or zero, if the number is zero). For strings, lists, and vectors, the inverse is defined as the reverse of the input.

```
(extend-type number
  IInvertible
  (invert [this] (if (zero? this) 0 (/ 1 this))))

(extend-type string
  IInvertible
  (invert [this] (apply str (reverse this))))

(extend-type List
  IInvertible
  (invert [this] (reverse this)))

(extend-type PersistentVector
  IInvertible
  (invert [this] (into [] (reverse this))))
```

If you load in this code, you can see that it works:

```
(proto.testproto/invert "abc")
;; => "cba"
(proto.testproto/invert 25)
;; => 0.04
(proto.testproto/invert 0)
;; => 0
```

```
(proto.testproto/invert '(1 2 3))
;; => (3 2 1)
(proto.testproto/invert [1 2 3])
;; => [3 2 1]
```

Admittedly, this is a somewhat contrived example. In the next section you will see how to extend an existing type.

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
    ))
```

Thus, the previous example could have been written equally well this way:

```
(extend-protocol IInvertible
  number
  (invert [this] (if (zero? this) 0 (/ 1 this)))

  string
  (invert [this] (apply str (reverse this)))

  List
  (invert [this] (reverse this))

  PersistentVector
  (invert [this] (into [] (reverse this))))
```

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 itself 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 look at a real life example.

In previous sections, we have explained the different kinds of built-in collections. For this example we will use a **set**. See this snippet 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 to use a regular expression as a 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 implement 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 a 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 determine at runtime if some object (instance of some type) satisfies the concrete protocol.

So, with the previous examples, if we check if a **`set`** instance satisfies an **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 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 the **`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 the **`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 an implementation for that multimethod.

### 3.10.3. Hierarchies

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

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 only from the one in which it is 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 allows 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 the 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 work 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. First, we define the multimethod with the **`defmulti`** form:

```
(defmulti stringify-shape
```

```
  "A function that prints a human readable representation
  of a shape keyword."
  identity
  :hierarchy #'h)
```

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

Second, 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.

Finally, if you give a keyword that isn't part of the hierarchy, you get the **`:default`** implementation:

```
(stringify-shape :triangle)
;; => "Unexpected object"
```

# 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 **data types**.

A data type 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"))
```

Its fields can be accessed using the prefix dot notation:

```
(.-firstname person)
```

```
;; => "Triss"
```

Types defined with **deftype** (and **defrecord**, which we will see later) create a host-backed class-like object associated with the current namespace. For convenience, *ClojureScript* also defines a constructor function called **→User** that can be imported 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))
```

We use this 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 the 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 data type 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’s 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 them:

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

And like maps, they 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.

Another 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 is an *ad hoc constructor* you can use to create objects without pre-defining a type. Protocol implementations are supplied the same as **deftype** and **defrecord**, but in contrast, **reify** 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"
```

## 3.11.5. Specify

**specify!** is an advanced alternative to **reify**, allowing you to add protocol implementations to an existing JavaScript object. This can be useful if you want to graft protocols onto a JavaScript library's components.

```
(def obj #js {})

(specify! obj
  IUser
  (full-name [_]
    "my full name"))

(full-name obj)
;; => "my full name"
```

**specify** is an immutable version of **specify!** that can be used on immutable, copyable values implementing **ICloneable** (e.g. ClojureScript collections).

```
(def a {})

(def b (specify a
         IUser
         (full-name [_]
           "my full name")))
```

```
(full-name a)
;; Error: No protocol method IUser.full-name defined for type cljs.core/
PersistentArrayMap: {}

(full-name b)
;; => "my full name"
```

## 3.12. Host interoperability

*ClojureScript*, in the same way as its 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 preceding sections of this chapter.

### 3.12.2. Interacting with platform types

*ClojureScript* comes with a little set of special forms that allows it to 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 any 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
```

You can invoke instance methods on JavaScript objects. The first example follows the pattern you have seen; the last one is a shortcut:

```
(.sqrt js/Math 2)
;; => 1.4142135623730951
(js/Math.sqrt 2)
;; => 1.4142135623730951
```

## Access to object properties

Access to an object's 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
```

## Property access shorthand

Symbols with the `js/` prefix can contain dots to denote nested property access. Both of the following expressions invoke the same function:

```
(.log js/console "Hello World")

(js/console.log "Hello World")
```

And both of the following expressions access the same property:

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

js/Math.PI
```

```
;; => 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 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 the previous section, you can also access 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 do not make recursive transformations, so if you have nested objects, the nested objects will not be converted. Consider this example that uses Clojurescript maps, then a similar one with JavaScript objects:

```
(def clj-map {:country {:code "FR" :name "France"}})
;; => {:country {:code "FR", :name "France"}}
(:code (:country clj-map))
;; => "FR"

(def js-obj #js {:country {:code "FR" :name "France"}})
;; => #js {:country {:code "FR", :name "France"}
(.-country js-obj)
;; => {:code "FR", :name "France"}
(.-code (.-country js-obj))
;; => nil
```

To solve that use case, *ClojureScript* comes with the **`clj→js`** and **`js→clj`** functions that transform Clojure collection types into JavaScript and back. Note that the conversion to ClojureScript changes the **`:country`** keyword to a string.

```
(clj->js {:foo {:bar "baz"}})
;; => #js {:foo #js {:bar "baz"}}
(js->clj #js {:country {:code "FR" :name "France"}}))
;; => {"country" {:code "FR", :name "France"}}
```

In the 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 the previous example, we saw 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 in 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 an indexed way to get its values:

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

In JavaScript, array index access is equivalent to object property access, 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. 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`**.

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

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

(s 1)
;; => 1

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

## 3.14. State management

We've learned that one of ClojureScript's fundamental ideas is immutability. Both scalar values and collections are immutable in ClojureScript, except those mutable types present in the JS host like **`Date`**.

Immutability has many great properties but we are sometimes faced with the need to model values that change over time. How can we achieve this if we can't change data structures in place?

### 3.14.1. Vars

Vars can be redefined at will inside a namespace but there is no way to know **when** they change. The inability to redefine vars from other namespaces is a bit limiting; also, if we are modifying state, we're probably interested in knowing when it occurs.

### 3.14.2. Atoms

ClojureScript gives us the **`Atom`** type, which is an object containing a value that can be altered at will. Besides altering its value, it also supports observation through watcher functions that can be attached and detached from it and validation for ensuring that the value contained in the atom is always valid.

If we were to model an identity corresponding to a person called Ciri, we could wrap an immutable value containing Ciri's data in an atom. Note that we can get the atom's value with the **deref** function or using its shorthand **@** notation:

```
(def ciri (atom {:name "Cirilla" :lastname "Fiona" :age 20}))
;; #<Atom: {:name "Cirilla", :lastname "Fiona", :age 20}>

(deref ciri)
;; {:name "Cirilla", :lastname "Fiona", :age 20}

@ciri
;; {:name "Cirilla", :lastname "Fiona", :age 20}
```

We can use the **swap!** function on an atom to alter its value with a function. Since Ciri's birthday is today, let's increment her age count:

```
(swap! ciri update :age inc)
;; {:name "Cirilla", :lastname "Fiona", :age 21}

@ciri
;; {:name "Cirilla", :lastname "Fiona", :age 21}
```

The **reset!** functions replaces the value contained in the atom with a new one:

```
(reset! ciri {:name "Cirilla", :lastname "Fiona", :age 22})
;; {:name "Cirilla", :lastname "Fiona", :age 22}

@ciri
;; {:name "Cirilla", :lastname "Fiona", :age 22}
```

## Observation

We can add and remove watcher functions for atoms. Whenever the atom's value is changed through a **swap!** or **reset!**, all the atom's watcher functions will be called. Watchers are added with the **add-watch** function. Notice that each watcher has a key associated (**:logger** in the example) to it which is later used to remove the watch from the atom.

```
(def a (atom))

(add-watch a :logger (fn [key the-atom old-value new-value]
                       (println "Key:" key "Old:" old-value "New:" new-
value)))

(reset! a 42)
;; Key: :logger Old: nil New: 42
;; => 42

(swap! a inc)
;; Key: :logger Old: 42 New: 43
;; => 4

(remove-watch a :logger)
```

### 3.14.3. Volatiles

Volatiles, like atoms, are objects containing a value that can be altered. However, they don't provide the observation and validation capabilities that atoms provide. This makes them slightly more performant and a more suitable mutable container to use inside stateful functions that don't need observation nor validation.

Their API closely resembles that of atoms. They can be dereferenced to grab the value they contain and support swapping and resetting with **vswap!** and **vreset!** respectively:

```
(def ciri (volatile! {:name "Cirilla" :lastname "Fiona" :age 20}))
;; #<Volatile: {:name "Cirilla", :lastname "Fiona", :age 20}>

(volatile? ciri)
;; => true

(deref ciri)
;; {:name "Cirilla", :lastname "Fiona", :age 20}
```

```
(vswap! ciri update :age inc)
;; {:name "Cirilla", :lastname "Fiona", :age 21}

(vreset! ciri {:name "Cirilla", :lastname "Fiona", :age 22})
;; {:name "Cirilla", :lastname "Fiona", :age 22}
```

Note that another difference with atoms is that the constructor of volatiles uses a bang at the end. You create volatiles with **`volatile!`** and atoms with **`atom`**.

# 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 the other.

## 4.1. Getting Started with the 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 a great number of different environments where it can be executed - each with its own peculiarities.

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[1]

[1] http://leiningen.org/

cljsbuild[2] or boot[3]).

### 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[4]/ iojs[5].

There are others, such as Rhino (JDK 6+), Nashorn (JDK 8), 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. To use it, you should have jdk8 installed. *ClojureScript* itself only requires JDK 7, but the standalone compiler that we are going to use in this chapter requires JDK 8, which can be found at http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

You can download the latest *ClojureScript* compiler using **`wget`**:

```
wget https://github.com/clojure/clojurescript/releases/download/r1.7.28/
cljs.jar
```

The *ClojureScript* compiler is packaged in a standalone executable jar file, so this is the only file (along with JDK 8) that you need to compile your *ClojureScript* source code to JavaScript.

### 4.1.3. Compile for nodejs/iojs

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

2 https://github.com/emezeske/lein-cljsbuild
3 http://boot-clj.com/
4 https://nodejs.org/
5 https://iojs.org/en/index.html

There are different ways to install iojs, but the recommended way is using nvm (node version manager). You can read the instructions on how to install and use nvm on its home page[6].

When you have installed nvm, follow installing the latest version of iojs:

```
nvm install iojs-v2.5.0
nvm alias default iojs-v2.5.0
```

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

```
$ iojs --version
v2.5.0
```

## 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.

Start by creating the directory tree structure for our “hello world” application:

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

Resulting in this directory tree:

```
myapp
└── src
    └── myapp
        └── core.cljs
```

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

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

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

```
(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 tells the *ClojureScript* compiler the source directory and the output file. *ClojureScript* has a lot of other options, but at this moment we can ignore that.

Let's create the *myapp/build.clj* file with the following content:

```
(require '[cljs.build.api :as b])

(b/build "src"
 {:main 'myapp.core
  :output-to "main.js"
  :output-dir "out"
  :target :nodejs
  :verbose true})
```

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 to 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:

```
cd myapp
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 an application similar to the "hello world" example from the previous section to run in the browser environment. The minimal requirement for this application is just a browser that can execute JavaScript.

The process is almost the same, and the directory structure is the same. The only things that changes is the entry point of the application and the build script. So, start re-creating the directory tree from previous example in a different directory.

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

Resulting in this directory tree:

```
mywebapp
└── src
    └── mywebapp
        └── core.cljs
```

Then, write new content to the **`mywebapp/src/mywebapp/core.cljs`** file:

```
(ns mywebapp.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 *mywebapp/build.clj* file with the following content:

```
(require '[cljs.build.api :as b])

(b/build "src"
 {:output-to "main.js"
  :output-dir "out/"
  :source-map "main.js.map"
  :main 'mywebapp.core
  :verbose true
  :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 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 file sources used in a compilation. It is just for making source maps work properly with the rest of the code, not only your source.
- **`:optimizations`** indicates the compilation optimization. There are different values for this option, but that will be covered in subsequent sections in more detail.

To run the compilation, just execute the following command:

```
cd mywebapp;
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 the 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 *mywebapp* 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. The "Hello world!" message should appear there.

### 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 performs an incremental compilation.

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

```
(require '[cljs.build.api :as b])
```

```
(b/watch "src"
 {:output-to "main.js"
  :output-dir "out/"
  :source-map "main.js.map"
  :main 'mywebapp.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 src/mywebapp/core.cljs
Compiling out/cljs/core.cljs
Using cached cljs.core out/cljs/core.cljs
... done. Elapsed 0.754487937 seconds
Watching paths: /home/niwi/cljsbook/playground/mywebapp/src
```

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

### 4.1.6. Optimization levels

The *ClojureScript* compiler has different levels of optimization. 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 this phase.
2. Then, the *ClojureScript* compiler emits JavaScript code. The result 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

The advanced compilation level includes all transformations from simple optimization and additionally performs more aggressive optimizations and dead code elimination. This results in significantly smaller output files.

The **`:advanced`** optimizations can only work for a strict subset of JavaScript code that follows the Google Closure Compiler rules. *ClojureScript* generates valid JavaScript within this strict subset, but if you are interacting with third party JavaScript code, some additional tasks should be done to make everything work as expected.

This interaction with third party javascript libraries will be explained in later sections.

## 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.

*ClojureScript* comes with support for executing the REPL in different execution environments, each one has its own advantages or disadvantages. For example, you can run a REPL in nodejs but in that environment you don't have access to the DOM.

In summary: everything really depends on your needs or requirements.

### 4.2.2. Nashorn REPL

The Nashorn REPL is the easiest and maybe most painless REPL environment because it does not require any special stuff, just the JVM (JDK 8) that you have used in previous examples for running the *ClojureScript* compiler.

Let's start creating the *repl.clj* file with the following content:

```
(require '[cljs.repl]
         '[cljs.repl.nashorn])

(cljs.repl/repl
 (cljs.repl.nashorn/repl-env)
```

```
  :output-dir "out"
  :cache-analysis true)
```

Then, execute the following command to get the REPL up and running:

```
$ java -cp cljs.jar:src clojure.main repl.clj
To quit, type: :cljs/quit
cljs.user=> (+ 1 2)
3
```

You may have noticed that the REPL does not have support for history and other shell-like facilities. This is because the default REPL does not comes with "readline" support. But this problem can be solved using the simple tool called **`rlwrap`** that you can certainly find with the package manager of your operating system.

The **`rlwrap`** makes the REPL "readline" capable and will allow you to have command history, code navigation, and other shell-like utilities that will make your REPL experience much more pleasant. To use it, just prepend it to the previous command that we have used to execute the REPL:

```
$ rlwrap java -cp cljs.jar:src clojure.main repl.clj
To quit, type: :cljs/quit
cljs.user=> (+ 1 2)
3
```

### 4.2.3. Node REPL

This REPL uses nodejs/iojs as an execution environment and obviously requires that nodejs or iojs will be installed in your system.

Surely you are asking yourself, why do I need a repl with nodejs when we have nashorn that does not require any additional dependencies. The answer is very simple: node/iojs is the most used JavaScript execution environment on the backend, and it has a great amount of community packages around it.

Well, the good news is that starting a nodejs/iojs repl is very easy once you have it installed in your system. Start writing this content to a new **`repl.clj`** file:

```
(require '[cljs.repl]
         '[cljs.repl.node])

(cljs.repl/repl
 (cljs.repl.node/repl-env)
 :output-dir "out"
 :cache-analysis true)
```

And start the REPL like you have done it previously with nashorn REPL:

```
$ rlwrap java -cp cljs.jar:src clojure.main repl.clj
To quit, type: :cljs/quit
cljs.user=> (+ 1 2)
3
```

### 4.2.4. Browser REPL

This REPL is the most laborious to get up and running. This is because it uses a browser for its execution environment and it has additional requirements.

Let's start by creating a file named **`brepl.clj`** with the following content:

```
(require
  '[cljs.build.api :as b]
  '[cljs.repl :as repl]
  '[cljs.repl.browser :as browser])

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

(repl/repl (browser/repl-env)
  :output-dir "out")
```

This script builds the source, just as we did earlier, and then starts the REPL.

But the browser REPL also requires that some code be executed in the browser before getting the REPL working. To do that, just re-create the application structure very similar to the one that we have used in previous sections:

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

Then, write new content to the **`src/myapp/core.cljs`** file:

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

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

(enable-console-print!)

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

And finally, create the missing *index.html* file that is going to be used as the entry point for running the browser side code of the REPL:

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

Well, that was a lot of setup! But trust us, it's all worth it when you see it in action. To do that, just execute the **`brepl.clj`** in the same way that we have done it in previous examples:

```
$ rlwrap java -cp cljs.jar:src clojure.main brepl.clj
```

```
Compiling client js ...
Waiting for browser to connect ...
```

And finally, open your favourite browser and go to http://localhost:9000/. Once the page is loaded (the page will be blank), switch back to the console where you have run the REPL and you will see that it is up and running:

```
[...]
To quit, type: :cljs/quit
cljs.user=> (+ 14 28)
42
```

One of the big advantages of this REPL is that you have access to all the browser environment. To demonstrate it, just enter **`(js/alert "hello world")`** in the repl. The result of that will be the typical browser alert dialog.

## 4.3. The Closure Library

The Google Closure Library is a javascript library developed by Google, based on a modular architecture and provides cross-browser functions for DOM manipulations and events, ajax and JSON, among other features.

It's written specifically to take advantage of the Closure Compiler (that is used internally by the *ClojureScript* compiler).

And *ClojureScript* is built on Closure Compiler and Closure Library. In fact, *ClojureScript* namespaces are Closure modules. This means that you can interact with the Closure Library in a very easy way:

```
(ns yourapp.core
  (:require [goog.dom :as dom]))

(def element (dom/getElement "body"))
```

With the previous snippet of code you can observe the way you can import the **dom** module of the closure library and use one function declared in that module.

Additionally, the closure library also exposes "special" modules that behave like a class or object. For importing these kinds of things you should use the **`:import`** directive.

```
(ns yourapp.core
  (:import goog.History))

(def instance (History.))
```

If you are familiar with Clojure, it imports Java classes using the same **`:import`** directive. However, if you define types (classes) using *ClojureScript* primitives, you should not use **`:import`** to import them, the standard **`:require`** directives should be used.

## 4.4. Dependency management

Until now, we have used the builtin *ClojureScript* toolchain to compile our sources to JavaScript. This is the minimal setup required for working with and understanding the compiler, but for larger projects, we often require a larger build tool that can manage a project's dependencies on other libraries.

Thus, this chapter intends to explain how you can use **Leiningen**, the de facto clojure build and dependency management tool, to build *ClojureScript* projects. There is another build tool called **boot** that is growing in popularity, but at this moment it will not be covered in this book.

### 4.4.1. Installing leiningen

The installation process of leiningen is quite simple; just follow these steps:

```
mkdir ~/bin
cd ~/bin
wget https://raw.githubusercontent.com/technomancy/leiningen/stable/bin/
lein
chmod a+x ./lein
export PATH=$PATH:~/bin
```

Make sure that the **`~/bin`** directory is always set on your path. To make it permanent, add the line starting with **`export`** to your **`~/.bashrc`** file (we are supposing that you are using the bash shell).

Now, open another clean terminal and execute **`lein version`**:

```
$ lein version
Leiningen 2.5.1 on Java 1.8.0_45 OpenJDK 64-Bit Server VM
```

> We are supposing that you are using a Unix-like system such as Linux or BSD. If you are a Windows user, please check the instructions on the Leiningen homepage[7]. You can also get the Linux/Mac OSX/BSD version of the leiningen script at the web site.

### 4.4.2. First project

The best way to show how a tool works is by creating a toy project with it. In this case, we will create a small application that calculates if a year is a leap year or not. To start, we will use the **mies** leiningen template.

> Templates are a facility in leiningen for creating an initial project structure. The clojure community has a great many of them. In this case we'll use the **mies** template that was started by the clojurescript core developer. Consult the leiningen docs to learn more about templates.

Let's start creating the project layout:

```
$ lein new mies leapyears
$ cd leapyears # move into newly created project directory
```

The project has the following structure:

```
leapyears
├── index.html
├── project.clj
```

[7] http://leiningen.org/

```
├── README.md
├── scripts
│   ├── build
│   ├── release
│   ├── watch
│   ├── repl
│   └── brepl
└── src
    └── leapyears
        └── core.cljs
```

The **project.clj** file contains information that Leiningen uses to download dependencies and build the project. For now, just trust that everything in that file is exactly as it should be.

Open the **index.html** file and add the following content at the beginning of body:

```
<section class="viewport">
  <div id="result">
    ----
  </div>
  <form action="" method="">
    <label for="year">Input a year</label>
    <input id="year" name="year" />
  </form>
</section>
```

And the next step is adding some code to make the form interactive. Put the following code into the **src/leapyear/core.cljs**:

```
(ns leapyears.core
  (:require [goog.dom :as dom]
            [goog.events :as events]
            [cljs.reader :refer (read-string)]))

(enable-console-print!)

(def input (dom/getElement "year"))
(def result (dom/getElement "result"))
```

```
(defn leap?
  [year]
  (and (zero? (js-mod year 4))
       (pos? (js-mod year 100))
       (pos? (js-mod year 400))))

(defn on-change
  [event]
  (let [target (.-target event)
        value (read-string (.-value target))]
    (if (leap? value)
      (set! (.-innerHTML result) "YES")
      (set! (.-innerHTML result) "NO"))))

(events/listen input "keyup" on-change)
```

Now, compile the clojurescript code with:

```
$ ./scripts/watch
```

This script executes the following behind the scenes, similar to the **`java`** build commands from the previous section, but with our **`lein`** build tool:

```
rlwrap lein trampoline run -m clojure.main scripts/watch.clj
```

> you should have **`rlwrap`** installed on your system.

Finally, open the **`index.html`** file in a browser. Typing a year in the textbox should display an indication of its leap year status.

You may have noticed other files in the scripts directory, like **`build`** and **`release`**. These are the same build scripts mentioned in the previous section, but we will stick with **`watch`** here.

### 4.4.3. Managing dependencies

The real purpose of using Leiningen for the ClojureScript compilation process is to automate the retrieval of dependencies. This is dramatically simpler than retrieving them manually.

The dependencies, among other parameters, are declared in the **`project.clj`** file and have this form (from **mies** template):

```
(defproject leapyears "0.1.0-SNAPSHOT"
  :description "FIXME: write this!"
  :url "http://example.com/FIXME"
  :dependencies [[org.clojure/clojure "1.7.0"]
                 [org.clojure/clojurescript "0.0-3308" :classifier "aot"
                  :exclusion [org.clojure/data.json]]
                 [org.clojure/data.json "0.2.6" :classifier "aot"]]
  :jvm-opts ^:replace ["-Xmx1g" "-server"]
  :node-dependencies [[source-map-support "0.3.2"]]
  :plugins [[lein-npm "0.5.0"]]
  :source-paths ["src" "target/classes"]
  :clean-targets ["out" "release"]
  :target-path "target")
```

And here is a brief explanation of the properties relevant for ClojureScript:

- **`:dependencies`**: a vector of dependencies that your project needs.
- **`:clean-targets`**: a vector of paths that **`lein clean`** should delete.

The dependencies in ClojureScript are packaged using **`jar`** files. If you are coming from Clojure or any JVM language, **`jar`** files will be very familiar to you. But if you aren't familiar with them, do not worry: a .jar file is like a plain zip file that contains the **`project.clj`** for the library, some metadata, and the ClojureScript sources. The packaging will be explained in another section.

Clojure packages are often published on Clojars[8]; also you can find many different third party libraries on the ClojureScript Wiki[9].

[8] http://clojars.org

To understand dependencies better, we are going to replace the DOM utiltiles of the Google Closure library with a third party library for interacting with the DOM.

Let's start by adding **`[prismatic/dommy "1.1.0"]`** at the end of the **`:dependencies`** vector in **project.clj**:

```
:dependencies [[org.clojure/clojure "1.7.0"]
               [org.clojure/clojurescript "0.0-3308" :classifier "aot"
                :exclusion [org.clojure/data.json]]
               [org.clojure/data.json "0.2.6" :classifier "aot"]
               [prismatic/dommy "1.1.0"]]
```

At this moment you should restart the **`watch`** process that we had started at the beginning of this subchapter:

```
$ ./scripts/watch
Retrieving prismatic/dommy/1.1.0/dommy-1.1.0.pom from clojars
Retrieving prismatic/dommy/1.1.0/dommy-1.1.0.jar from clojars
Building ...
... done. Elapsed 0.227999601 seconds
Watching paths: /home/niwi/cljsbook/playground/leapyears/src
```

You can observe that the dommy dependency is successfully downloaded, and then the watch process is started. In fact, Leiningen will ensure all dependencies are downloaded before performing any tasks.

Now it's time to write some code. Replace the contents of the **`src/leapyear/core.cljs`** with the following content:

```
(ns leapyears.core
  (:require [dommy.core :as dom]
            [cljs.reader :refer (read-string)]))

(enable-console-print!)

(def input (dom/sel1 :#year))
```

[9] https://github.com/clojure/clojurescript/wiki#libraries

```
(def result (dom/sel1 :#result))

(defn leap?
  [year]
  (and (zero? (js-mod year 4))
       (pos? (js-mod year 100))
       (pos? (js-mod year 400))))

(defn on-change
  [event]
  (let [target (.-target event)
        value (read-string (dom/value target))]
    (if (leap? value)
      (dom/set-html! result "YES")
      (dom/set-html! result "NO"))))

(dom/listen! input :keyup on-change)
```

As you can observe, the code now looks more “Clojure-friendly.” The host interoperabilty calls have now been replaced by functions provided by this third-party library.

## 4.5. Using javascript libraries

TBD

## 4.6. Unit testing

As you may expect, testing in *ClojureScript* consists of the same concepts as you may know from any other language such as Clojure, Java, Python, JavaScript, etc.

Regardless of the language, the main objective of unit testing is to run some test code, and verify that your code behaves as expected and returns without raising unexpected exceptions.

The immutablity of *ClojureScript* in fact helps your program to be less error prone and facilitates testing a little bit. Another advantage of *ClojureScript* is that it tends to use plain data instead of complex objects. Building "mock" objects for testing is greatly simplified.

### 4.6.1. First steps

The "official" *ClojureScript* testing framework is in the core library under the "cljs.test" namespace. It is a very simple library, but it should be more than enough for our purposes.

There are other libraries that offer additional features or directly different approaches for testing, such as test.check[10]. However, they will not be covered in this subchapter.

Start creating a new project using the **mies** leiningen template for experimenting with tests:

```
$ lein new mies appfortesting
$ cd appfortesting
```

This project will contain the same layout as we have seen in the **dependency management** subchapter, so we won’t explain it again.

The next step is a creating a directory tree for our tests:

```
$ mkdir -p test/appfortesting
$ touch test/appfortesting/core_tests.cljs
```

Also, we should adapt the existing `watch.clj` script to work with this newly created test directory:

```
(require '[cljs.build.api :as b])

(b/watch (b/inputs "test" "src")
  {:main 'appfortesting.core_tests
   :target :nodejs
   :output-to "out/appfortesting.js"
   :output-dir "out"
   :verbose true})
```

[10] https://github.com/clojure/test.check

This new script will compile and watch both directories "src" and "test", and sets the new entry point to the **`appfortesting.core_tests`** namespace.

Next, write some test code into the **`core_tests.cljs`** file:

```
(ns appfortesting.core-tests
  (:require [cljs.test :as t]))

(enable-console-print!)

(t/deftest my-first-test
  (t/is (= 1 2)))

(set! *main-cli-fn* #(t/run-tests))
```

The relevant part of that code snippet is:

```
(t/deftest my-first-test
  (t/is (= 1 2)))
```

The **`deftest`** macro is a basic primitive for defining our tests. It takes a name as its first parameter, followed by one or multiple "asserts" using the **`is`** macro. In that example, the assert tries check that **`(= 1 2)`** is true.

Let’s try to run this. First start the watch process:

```
$ ./scripts/watch
Building ...
Analyzing jar:file:/home/niwi/.m2/repository/org/clojure/
clojurescript/0.0-3308/clojurescript-0.0-3308-aot.jar!/cljs/core.cljs
Compiling out/cljs/core.cljs
Using cached cljs.core out/cljs/core.cljs
... done. Elapsed 3.862126827 seconds
Watching paths: /home/niwi/cljsbook/playground/appfortesting/test, /home/
niwi/cljsbook/playground/appfortesting/src
```

And when the compilation is finished, try to run the compiled file with **`iojs`** (or **`node`**):

```
$ iojs out/appfortesting.js
```

```
Testing appfortesting.core-tests

FAIL in (my-first-test) (cljs/test.js:374:14)
expected: (= 1 2)
  actual: (not (= 1 2))

Ran 1 tests containing 1 assertions.
1 failures, 0 errors.
```

You can observe that the expected assert failure is successfully printed in the console. To fix the test, just change the **=** with **not=** and run the file again:

```
$ iojs out/appfortesting.js

Testing appfortesting.core-tests

Ran 1 tests containing 1 assertions.
0 failures, 0 errors.
```

It is fine to test these kinds of assertions, but they are not very useful. Let's go to test some application code. For this, we will use a function to check if a year is a leap year or not. Then, write the following content to the **src/appfortesting/core.clj** file:

```
(defn leap?
  [year]
  (and (zero? (js-mod year 4))
       (pos? (js-mod year 100))
       (pos? (js-mod year 400))))
```

Next, write a new test case to check that our new function defined in the **appfortesting.core** namespace works properly. This is the new content of the **core_tests.cljs** file:

```
(ns appfortesting.core-tests
  (:require [cljs.test :as t]
            [appfortesting.core :as core]))

(enable-console-print!)
```

```
(t/deftest my-first-test
  (t/is (not= 1 2)))

(t/deftest my-second-test
  (t/is (core/leap? 1980))
  (t/is (not (core/leap? 1981))))

(set! *main-cli-fn* #(t/run-tests))
```

Run the compiled file again to see that there are now two tests running. The first test fails as it did before, but our leap year test passes as expected.

### 4.6.2. Async Testing

One of the peculiarities of *ClojureScript* is that it runs on asynchronous, single-threaded execution environments, and this has its own challenges.

In async execution environments, we should be able to test asynchronous functions. To this end, the *ClojureScript* testing library offers the **`async`** macro, allowing you to create tests that play well with asynchronous code.

First, we need to write a function that works in an asynchronous way. For this purpose, we will create the **`async-leap?`** predicate that will do the same operation but asychronously return a result using a callback:

```
(defn async-leap?
  [year callback]
  (js/setImmediate
   (fn []
     (let [result (and (zero? (js-mod year 4))
                       (pos? (js-mod year 100))
                       (pos? (js-mod year 400)))]
       (callback result)))))
```

The JavaScript function **`setImmediate`** is used to emulate an asynchronous task, and the callback is executed with the result of that predicate.

To test it, we should write a test case using the previously mentioned **`async`** macro:

```
(t/deftest my-async-test
  (t/async done
    (core/async-leap? 1980 (fn [result]
                             (t/is (true? result))
                             (done)))))
```

The **done** function exposed by the **async** macro should be called after the asynchronous operation is finished and all assertions have run.

It is very important to execute the **done** function only once. Omitting or executing it twice may cause strange behavior and should be avoided.

### 4.6.3. Fixtures

TBD

### 4.6.4. Integrating with CI

Most continuous integration tools and services expect that test scripts you provide return a standard exit code. But the ClojureScript test framework cannot customize this exit code without some configuration, because JavaScript lacks a universal exit code API for ClojureScript to use.

To fix this, the *ClojureScript* test framework provides an avenue for executing custom code after the tests are done. This is where you are expected to set the environment-specific exit code depending on the final test status: **0** for success, **1** for failure.

Insert this code at the end of **core_tests.cljs**:

```
(defmethod t/report [::t/default :end-run-tests]
  [m]
  (if (t/successful? m)
    (set! (.-exitCode js/process) 0)
    (set! (.-exitCode js/process) 1)))
```

Now, you may check the exit code of the test script after running:

```
$ iojs out/appfortesting.js
```

```
$ echo $?
```

This code snippet obviously assumes that you are running the tests using **iojs** or **nodejs**. If you are running your script in another execution environment, you should be aware of how you can set the exit code in that environment and modify the previous snippet accordingly.

# Chapter 5. Language (advanced topics)

This chapter intends to explain some advanced topics that are part of the language and that does not fits in the first chapter. The good candidates for this section are transducers, core protocols, transients, metadata. In summary: topics that are not mandatory for understand the language.

## 5.1. Transducers

### 5.1.1. Data transformation

ClojureScript offers a rich vocabulary for data transformation in terms of the sequence abstraction, which makes such transformations highly general and composable. Let's see how we can combine several collection processing functions to build new ones. We will be using a simple problem throughout this section: splitting grape clusters, filtering out the rotten ones, and cleaning them. We have a collection of grape clusters like the following:

```
(def grape-clusters [{:grapes [{:rotten? false :clean? false}
                               {:rotten? true :clean? false}]
                     :color :green}
                    {:grapes [{:rotten? true :clean? false}
                              {:rotten? false :clean? false}]
                     :color :black}])
```

We are interested in splitting the grape clusters into individual grapes, discarding the rotten ones and cleaning the remaining grapes so they are ready for eating. We are well-equipped in ClojureScript for this data transformation task; we could implement it using the familiar **`map`**, **`filter`** and **`mapcat`** functions:

```
(defn split-cluster
  [c]
  (:grapes c))

(defn not-rotten
  [g]
```

```
  (not (:rotten? g)))

(defn clean-grape
  [g]
  (assoc g :clean? true))

(->> grape-clusters
     (mapcat split-cluster)
     (filter not-rotten)
     (map clean-grape))
;; => ({rotten? false :clean? true} {:rotten? false :clean? true})
```

In the above example we succintly solved the problem of selecting and cleaning the grapes, and we can even abstract such transformations by combining the **`mapcat`**, **`filter`** and **`map`** operations using partial application and function composition:

```
(def process-clusters
  (comp
    (partial map clean-grape)
    (partial filter not-rotten)
    (partial mapcat split-cluster)))

(process-clusters grape-clusters)
;; => ({rotten? false :clean? true} {:rotten? false :clean? true})
```

The code is very clean, but it has a few problems. For example, each call to **`map`**, **`filter`** and **`mapcat`** consumes and produces a sequence that, although lazy, generates intermediate results that will be discarded. Each sequence is fed to the next step, which also returns a sequence. Wouldn't be great if we could do the transformation in a single transversal of the **`grape-cluster`** collection?

Another problem is that even though our **`process-clusters`** function works with any sequence, we can't reuse it with anything that is not a sequence. Imagine that instead of having the grape cluster collection available in memory it is being pushed to us asynchronously in a stream. In that situation we couldn't reuse **`process-clusters`** since usually **`map`**, **`filter`** and **`mapcat`** have concrete implementations depending on the type.

### 5.1.2. Generalizing to process transformations

The process of mapping, filtering or mapcatting isn't necesarily tied to a concrete type, but we keep reimplementing them for different types. Let's see how we can generalize such processes to be context independent. We'll start by implementing naive versions of **`map`** and **`filter`** first to see how they work internally:

```
(defn my-map
  [f coll]
  (when-let [s (seq coll)]
    (cons (f (first s)) (my-map f (rest s)))))

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

(defn my-filter
  [pred coll]
  (when-let [s (seq coll)]
    (let [f (first s)
          r (rest s)]
      (if (pred f)
        (cons f (my-filter pred r))
        (my-filter pred r)))))

(my-filter odd? [0 1 2])
;; => (1)
```

As we can see, they both assume that they receive a seqable and return a sequence. Like many recursive functions they can be implemented in terms of the already familiar **`reduce`** function. Note that functions that are given to reduce receive an accumulator and an input and return the next accumulator. We'll call these types of functions reducing functions from now on.

```
(defn my-mapr
  [f coll]
  (reduce (fn [acc input]         ;; reducing function
            (conj acc (f input)))
          []                      ;; initial value
          coll))                  ;; collection to reduce
```

```
(my-mapr inc [0 1 2])
;; => [1 2 3]

(defn my-filterr
  [pred coll]
  (reduce (fn [acc input]         ;; reducing function
            (if (pred input)
              (conj acc input)
              acc))
          []                      ;; initial value
          coll))                  ;; collection to reduce

(my-filterr odd? [0 1 2])
;; => [1]
```

We've made the previous versions more general since using **reduce** makes our functions work on any thing that is reducible, not just sequences. However you may have noticed that, even though **my-mapr** and **my-filterr** don't know anything about the source (**coll**) they are still tied to the output they produce (a vector) both with the initial value of the reduce (**[]**) and the hardcoded **conj** operation in the body of the reducing function. We could have accumulated results in another data structure, for example a lazy sequence, but we'd have to rewrite the functions in order to do so.

How can we make these functions truly generic? They shouldn't know about either the source of inputs they are transforming nor the output that is generated. Have you noticed that **conj** is just another reducing function? It takes an accumulator and an input and returns another accumulator. So, if we parameterise the reducing function that **my-mapr** and **my-filterr** use, they won't know anything about the type of the result they are building. Let's give it a shot:

```
(defn my-mapt
  [f]                         ;; function to map over inputs
  (fn [rfn]                   ;; parameterised reducing function
    (fn [acc input]           ;; transformed reducing function, now it
 maps `f`!
      (rfn acc (f input)))))

(def incer (my-mapt inc))
```

```
(reduce (incer conj) [] [0 1 2])
;; => [1 2 3]

(defn my-filtert
  [pred]                        ;; predicate to filter out inputs
  (fn [rfn]                     ;; parameterised reducing function
    (fn [acc input]             ;; transformed reducing function, now it
 discards values based on `pred`!
      (if (pred input)
        (rfn acc input)
        acc))))

(def only-odds (my-filtert odd?))

(reduce (only-odds conj) [] [0 1 2])
;; => [1]
```

That's a lot of higher-order functions so let's break it down for a better understanding of what's going on. We'll examine how **`my-mapt`** works step by step. The mechanics are similar for **`my-filtert`**, so we'll leave it out for now.

First of all, **`my-mapt`** takes a mapping function; in the example we are giving it **`inc`** and getting another function back. Let's replace **`f`** with **`inc`** to see what we are building:

```
(def incer (my-mapt inc))
;; (fn [rfn]
;;   (fn [acc input]
;;     (rfn acc (inc input))))
;;               ^^^
```

The resulting function is still parameterised to receive a reducing function to which it will delegate, let's see what happens when we call it with **`conj`**:

```
(incer conj)
;; (fn [acc input]
;;   (conj acc (inc input)))
;;    ^^^^
```

We get back a reducing function which uses **inc** to transform the inputs and the **conj** reducing function to accumulate the results. In essence, we have defined map as the transformation of a reducing function. The functions that transform one reducing function into another are called transducers in ClojureScript.

To ilustrate the generality of transducers, let's use different sources and destinations in our call to **reduce**:

```
(reduce (incer str) "" [0 1 2])
;; => "123"

(reduce (only-odds str) "" '(0 1 2))
;; => "1"
```

The transducer versions of **map** and **filter** transform a process that carries inputs from a source to a destination but don't know anything about where the inputs come from and where they end up. In their implementation they contain the *essence* of what they accomplish, independent of context.

Now that we know more about transducers we can try to implement our own version of **mapcat**. We already have a fundamental piece of it: the **map** transducer. What **mapcat** does is map a function over an input and flatten the resulting structure one level. Let's try to implement the catenation part as a transducer:

```
(defn my-cat
  [rfn]
  (fn [acc input]
    (reduce rfn acc input)))

(reduce (my-cat conj) [] [[0 1 2] [3 4 5]])
;; => [0 1 2 3 4 5]
```

The **my-cat** transducer returns a reducing function that catenates its inputs into the accumulator. It does so reducing the **input** reducible with the **rfn** reducing function and using the accumulator (**acc**) as the initial value for such reduction. **mapcat** is simply the composition of **map** and **cat**. The order in which transducers are composed may seem backwards but it'll become clear in a moment.

```
(defn my-mapcat
  [f]
  (comp (my-mapt f) my-cat))

(defn dupe
  [x]
  [x x])

(def duper (my-mapcat dupe))

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

### 5.1.3. Transducers in ClojureScript core

Some of the ClojureScript core functions like **`map`**, **`filter`** and **`mapcat`** support an arity 1 version that returns a transducer. Let's revisit our definition of **`process-cluster`** and define it in terms of transducers:

```
(def process-clusters
  (comp
    (mapcat split-cluster)
    (filter not-rotten)
    (map clean-grape)))
```

A few things changed since our previous definition **`process-clusters`**. First of all, we are using the transducer-returning versions of **`mapcat`**, **`filter`** and **`map`** instead of partially applying them for working on sequences.

Also you may have noticed that the order in which they are composed is reversed, they appear in the order they are executed. Note that all **`map`**, **`filter`** and **`mapcat`** return a transducer. **`filter`** transforms the reducing function returned by **`map`**, applying the filtering before proceeding; **`mapcat`** transforms the reducing function returned by **`filter`**, applying the mapping and catenation before proceeding.

One of the powerful properties of transducers is that they are combined using regular function composition. What's even more elegant is that the composition of various transducers is itself a transducer! This means that our **`process-cluster`** is a transducer too, so we have defined a composable and context-independent algorithmic transformation.

Many of the core ClojureScript functions accept a transducer, let's look at some examples with our newly created **`process-cluster`**:

```
(into [] process-clusters grape-clusters)
;; => [{:rotten? false, :clean? true} {:rotten? false, :clean? true}]

(sequence process-clusters grape-clusters)
;; => ({:rotten? false, :clean? true} {:rotten? false, :clean? true})

(reduce (process-clusters conj) [] grape-clusters)
;; => [{:rotten? false, :clean? true} {:rotten? false, :clean? true}]
```

Since using **`reduce`** with the reducing function returned from a transducer is so common, there is a function for reducing with a transformation called **`transduce`**. We can now rewrite the previous call to **`reduce`** using **`transduce`**:

```
(transduce process-clusters conj [] grape-clusters)
;; => [{:rotten? false, :clean? true} {:rotten? false, :clean? true}]
```

### 5.1.4. Initialisation

In the last example we provided an initial value to the **`transduce`** function (**`[]`**) but we can omit it and get the same result:

```
(transduce process-clusters conj grape-clusters)
;; => [{:rotten? false, :clean? true} {:rotten? false, :clean? true}]
```

What's going on here? How can **`transduce`** know what initial value use as an accumulator when we haven't specified it? Try calling **`conj`** without any arguments and see what happens:

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

The **conj** function has a arity 0 version that returns an empty vector but is not the only reducing function that supports arity 0. Let's explore some others:

```
(+)
;; => 0

(*)
;; => 1

(str)
;; => ""

(= identity (comp))
;; => true
```

The reducing function returned by transducers must support the arity 0 as well, which will typically delegate to the transformed reducing function. There is no sensible implementation of the arity 0 for the transducers we have implemented so far, so we'll simply call the reducing function without arguments. Here's how our modified **my-mapt** could look like:

```
(defn my-mapt
  [f]
  (fn [rfn]
    (fn
      ([] (rfn))                ;; arity 0 that delegates to the reducing
 fn
      ([acc input]
        (rfn acc (f input))))))
```

The call to the arity 0 of the reducing function returned by a transducer will call the arity 0 version of every nested reducing function, eventually calling the outermost reducing function. Let's see an example with our already defined **process-clusters** transducer:

```
((process-clusters conj))
;; => []
```

The call to the arity 0 flows through the transducer stack, eventually calling **`(conj)`**.

### 5.1.5. Stateful transducers

So far we've only seen purely functional transducer; they don't have any implicit state and are very predictable. However, there are many data transformation functions that are inherently stateful, like **`take`**. **`take`** receives a number **`n`** of elements to keep and a collection and returns a collection with at most **`n`** elements.

```
(take 10 (range 100))
;; => (0 1 2 3 4 5 6 7 8 9)
```

Let's step back for a bit and learn about the early termination of the **`reduce`** function. We can wrap an accumulator in a type called **`reduced`** for telling **`reduce`** that the reduction process should terminate immediately. Let's see an example of a reduction that aggregates the inputs in a collection and finishes as soon as there are 10 elements in the accumulator:

```
(reduce (fn [acc input]
          (if (= (count acc) 10)
            (reduced acc)
            (conj acc input)))
         []
         (range 100))
;; => [0 1 2 3 4 5 6 7 8 9]
```

Since transducers are modifications of reducing functions they also use **`reduced`** for early termination. Note that stateful transducers may need to do some cleanup before the process terminates, so they must support an arity 1 as a "completion" step. Usually, like with arity 0, this arity will simply delegate to the transformed reducing function's arity 1.

Knowing this we are able to write stateful transducers like **`take`**. We'll be using mutable state internally for tracking the number of inputs we have seen so far, and wrap the accumulator in a **`reduced`** as soon as we've seen enough elements:

```
(defn my-take
```

```
  [n]
  (fn [rfn]
    (let [remaining (volatile! n)]
      (fn
        ([] (rfn))
        ([acc] (rfn acc))
        ([acc input]
          (let [rem @remaining
                nr (vswap! remaining dec)
                result (if (pos? rem)
                         (rfn acc input)   ;; we still have items to take
                         acc)]             ;; we're done, acc becomes the
result
            (if (not (pos? nr))
              (ensure-reduced result)      ;; wrap result in reduced if
not already
              result)))))))
```

This is a simplified version of the **take** function present in ClojureScript core. There are a few things to note here so let's break it up in pieces to understand it better.

The first thing to notice is that we are creating a mutable value inside the transducer. Note that we don't create it until we receive a reducing function to transform. If we created it before returning the transducer we couldn't use **my-take** more than once. Since the transducer is handed a reducing function to transform each time it is used, we can use it multiple times and the mutable variable will be created in every use.

```
(fn [rfn]
  (let [remaining (volatile! n)] ;; make sure to create mutable variables
 inside the transducer
    (fn
      ;; ...
)))

(def take-five (my-take 5))

(transduce take-five conj (range 100))
;; => [0 1 2 3 4]

(transduce take-five conj (range 100))
```

```
;; => [0 1 2 3 4]
```

Let's now dig into the reducing function returned from **my-take**. First of all we **deref** the volatile to get the number of elements that remain to be taken and decrement it to get the next remaining value. If there are still remaining items to take, we call **rfn** passing the accumulator and input; if not, we already have the final result.

```
([acc input]
  (let [rem @remaining
        nr (vswap! remaining dec)
        result (if (pos? rem)
                 (rfn acc input)
                 acc)]
    ;; ...
))
```

The body of **my-take** should be obvious by now. We check whether there are still items to be processed using the next remainder (**nr**) and, if not, wrap the result in a **reduced** using the **ensure-reduced** function. **ensure-reduced** will wrap the value in a **reduced** if it's not reduced already or simply return the value if it's already reduced. In case we are not done yet, we return the accumulated **result** for further processing.

```
(if (not (pos? nr))
  (ensure-reduced result)
  result)
```

We've seen an example of a stateful transducer but it didn't do anything in its completion step. Let's see an example of a transducer that uses the completion step to flush an accumulated value. We'll implement a simplified version of **partition-all**, which given a **n** number of elements converts the inputs in vectors of size **n**. For understanding its purpose better let's see what the arity 2 version gives us when providing a number and a collection:

```
(partition-all 3 (range 10))
;; => ((0 1 2) (3 4 5) (6 7 8) (9))
```

The transducer returning function of **`partition-all`** will take a number **`n`** and return a transducer that groups inputs in vectors of size **`n`**. In the completion step it will check if there is an accumulated result and, if so, add it to the result. Here's a simplified version of ClojureScript core **`partition-all`** function, where **`array-list`** is a wrapper for a mutable JavaScript array:

```
(defn my-partition-all
  [n]
  (fn [rfn]
    (let [a (array-list)]
      (fn
        ([] (rfn))
        ([result]
          (let [result (if (.isEmpty a)                  ;; no inputs
 accumulated, don't have to modify result
                         result
                         (let [v (vec (.toArray a))]
                           (.clear a)                    ;; flush array
 contents for garbage collection
                           (unreduced (rfn result v))))] ;; pass to `rfn`,
 removing the reduced wrapper if present
            (rfn result)))
        ([acc input]
          (.add a input)
          (if (== n (.size a))                           ;; got enough
 results for a chunk
            (let [v (vec (.toArray a))]
              (.clear a)
              (rfn acc v))                               ;; the
 accumulated chunk becomes input to `rfn`
            acc))))))

(def triples (my-partition-all 3))

(transduce triples conj (range 10))
;; => [[0 1 2] [3 4 5] [6 7 8] [9]]
```

## 5.1.6. Eductions

Eductions are a way to combine a collection and one or more transformations that can be reduced and iterated over, and that apply the transformations every time we do so. If we have a collection that we want to process and a transformation over it that we want others to extend, we can hand them a eduction, encapsulating the source collection and our transformation. We can create an eduction with the **`eduction`** function:

```
(def ed (eduction (filter odd?) (take 5) (range 100)))

(reduce + 0 ed)
;; => 25

(transduce (partition-all 2) conj ed)
;; => [[1 3] [5 7] [9]]
```

## 5.1.7. More transducers in ClojureScript core

We learned about **`map`**, **`filter`**, **`mapcat`**, **`take`** and **`partition-all`**, but there are a lot more transducers available in ClojureScript. Here is an incomplete list of some other intersting ones:

- **`drop`** is the dual of **`take`**, dropping up to **`n`** values before passing inputs to the reducing function
- **`distinct`** only allows inputs to occur once
- **`dedupe`** removes succesive duplicates in input values

We encourage you to explore ClojureScript core to see what other transducers are out there.

## 5.1.8. Defining our own transducers

Since most transducers are only interesting in the reducing step implementation and often delegate to the reducing function in arities 0 and 1, there is a little helper called **`completing`** that fills those arities for you:

```
;; if a completing function is not defined, identity will be used
(def my-transducer
  (completing
    (fn [acc input]
     ;; ...
     )))

(def my-transducer-with-completion
  (completing
    (fn [acc input]
     ;; ...
     )
    (fn [result]
     ;; ...
    )))
```

## 5.1.9. Transducible processes

A transducible process is any process that is defined in terms of a succession of steps ingesting input values. The source of input varies from one process to another. Most of our examples dealt with inputs from a collection or a lazy sequence, but it could be an asynchronous stream of values or a **`core.async`** channel. The outputs produced by each step are also different for every process; **`into`** creates a collection with every output of the transducer, **`sequence`** yields a lazy sequence, and asynchronous streams would probably push the outputs to their listeners.

In order to improve our understanding of transducible processes, we're going to implement an unbounded queue, since adding values to a queue can be thought in terms of a succession of steps ingesting input. First of all we'll define a protocol and a data type that implements the unbounded queue:

```
(defprotocol Queue
  (put! [q item] "put an item into the queue")
  (take! [q] "take an item from the queue")
  (shutdown! [q] "stop accepting puts in the queue"))

(deftype UnboundedQueue [^:mutable arr ^:mutable closed]
  Queue
```

```
  (put! [_ item]
    (assert (not closed))
    (assert (not (nil? item)))
    (.push arr item)
    item)
  (take! [_]
    (aget (.splice arr 0 1) 0))
  (shutdown! [_]
    (set! closed true)))
```

We defined the **`Queue`** protocol and as you may have noticed the implementation of **`UnboundedQueue`** doesn't know about transducers at all. It has a **`put!`** operation as its step function and we're going to implement the transducible process on top of that interface:

```
(defn unbounded-queue
  ([]
   (queue nil))
  ([xform]
   (let [put! (completing put!)
         xput! (if xform (xform put!) put!)
         q (UnboundedQueue. #js [] false)]
     (reify
       Queue
       (put! [_ item]
         (when-not (.-closed q)
           (let [val (xput! q item)]
             (if (reduced? val)
               (do
                 (xput! @val)  ;; call completion step
                 (shutdown! q) ;; respect reduced
                 @val)
               val))))
       (take! [_]
         (take! q))
       (shutdown! [_]
         (shutdown! q))))))
```

As you can see, the **`unbounded-queue`** constructor uses an **`UnboundedQueue`** instance internally, proxying the **`take!`** and **`shutdown!`** calls and implementing the transducible process logic in the **put!** function. Let's deconstruct it to understand what's going on.

```
(let [put! (completing put!)
      xput! (if xform (xform put!) put!)
      q (UnboundedQueue. #js [] false)]
  ;; ...
)
```

First of all, we use **completing** for adding the arity 0 and arity 1 to the **Queue** protocol's **put!** function. This will make it play nicely with transducers in case we give this reducing function to **xform** to derive another. After that, if a transducer (**xform**) was provided, we derive a reducing function applying the transducer to **put!**. If we're not handed a transducer we will just use **put!**. **q** is the internal instance of **UnboundedQueue**.

```
(reify
  Queue
  (put! [_ item]
    (when-not (.-closed q)
      (let [val (xput! q item)]
        (if (reduced? val)
          (do
            (xput! @val)  ;; call completion step
            (shutdown! q) ;; respect reduced
            @val)
          val))))
  ;; ...
)
```

The exposed **put!** operation will only be performed if the queue hasn't been shut down. Notice that the **put!** implementation of **UnboundedQueue** uses an assert to verify that we can still put values to it and we don't want to break that invariant. If the queue isn't closed we can put values into it, we use the possibly transformed **xput!** for doing so.

If the put operation gives back a reduced value it's telling us that we should terminate the transducible process. In this case that means shutting down the queue to not accept more values. If we didn't get a reduced value we can happily continue accepting puts.

Let's see how our queue behaves without transducers:

```
(def q (queue))
;; => #<[object Object]>

(put! q 1)
;; => 1
(put! q 2)
;; => 2

(take! q)
;; => 1
(take! q)
;; => 2
(take! q)
;; => nil
```

Pretty much what we expected, let's now try with a stateless transducer:

```
(def incq (queue (map inc)))
;; => #<[object Object]>

(put! incq 1)
;; => 2
(put! incq 2)
;; => 3

(take! incq)
;; => 2
(take! incq)
;; => 3
(take! incq)
;; => nil
```

To check that we've implemented the transducible process, let's use a stateful transducer. We'll use a transducer that will accept values while they aren't equal to 4 and will partition inputs in chunks of 2 elements:

```
(def xq (queue (comp
```

```
                  (take-while #(not= % 4))
                  (partition-all 2))))

(put! xq 1)
(put! xq 2)
;; => [1 2]
(put! xq 3)
(put! xq 4) ;; shouldn't accept more values from here on
(put! xq 5)
;; => nil

(take! xq)
;; => [1 2]
(take! xq) ;; seems like `partition-all` flushed correctly!
;; => [3]
(take! xq)
;; => nil
```

The example of the queue was heavily inspired by how **core.async** channels use transducers in their internal step. We'll discuss channels and their usage with transducers in a later section.

Transducible processes must respect **reduced** as a way for signaling early termination. For example, building a collection stops when encountering a **reduced** and **core.async** channels with transducers are closed. The **reduced** value must be unwrapped with **deref** and passed to the completion step, which must be called exactly once.

Transducible processes shouldn't expose the reducing function they generate when calling the transducer with their own step function since it may be stateful and unsafe to use from elsewhere.

## 5.2. Transients

Although ClojureScript's immutable and persistent data structures are reasonably performant there are situations in which we are transforming large data structures using multiple steps to only share the final result. For example, the core **into** function takes a collection and eagerly populates it with the contents of a sequence:

```
(into [] (range 100))
;; => [0 1 2 ... 98 99]
```

In the above example we are generating a vector of 100 elements **conj**-ing one at a time. Every intermediate vector that is not the final result won't be seen by anybody except the **into** function and the array copying required for persistence is an unnecesary overhead.

For these situations ClojureScript provides a special version of some of its persistent data structures, which are called transients. Maps, vectors and sets have a transient counterpart. Transients are always derived from a persistent data structure using the **transient** function, which creates a transient version in constant time:

```
(def tv (transient [1 2 3]))
;; => #<[object Object]>
```

Transients support the read API of their persistent counterparts:

```
(def tv (transient [1 2 3]))

(nth tv 0)
;; => 1

(get tv 2)
;; => 3

(def tm (transient {:language "ClojureScript"}))

(:language tm)
;; => "ClojureScript"

(def ts (transient #{:a :b :c}))

(contains? ts :a)
;; => true

(:a ts)
;; => :a
```

Since transients don't have persistent and immutable semantics for updates they can't be transformed using the already familiar **conj** or **assoc** functions. Instead, the transforming functions that work on transients end with a bang. Let's look at an example using **conj!** on a transient:

```
(def tv (transient [1 2 3]))

(conj! tv 4)
;; => #<[object Object]>

(nth tv 3)
;; => 4
```

As you can see, the transient version of the vector is neither immutable or persistent. Instead, the vector is mutated in place. Although we could transform **tv** repeatedly using **conj!** on it we shouldn't abandon the idioms used with the persistent data structures: when transforming a transient, use the returned version of it for further modifications like in the following example:

```
(-> [1 2 3]
  transient
  (conj! 4)
  (conj! 5))
;; => #<[object Object]>
```

We can convert a transient back to a persistent and immutable data structure by calling **persistent!** on it. This operation, like deriving a transient from a persistent data structure, is done in constant time.

```
(-> [1 2 3]
  transient
  (conj! 4)
  (conj! 5)
  persistent!)
;; => [1 2 3 4 5]
```

A peculiarity of transforming transients into persistent structures is that the transient version is invalidated after being converted to a persistent data structure and we can't do further transformations to it. This happens because the derived persistent data structure uses the transient's internal nodes and mutating them would break the immutability and persistent guarantees:

```
(def tm (transient {}))
;; => #<[object Object]>

(assoc! tm :foo :bar)
;; => #<[object Object]>

(persistent! tm)
;; => {:foo :bar}

(assoc! tm :baz :frob)
;; Error: assoc! after persistent!
```

Going back to our initial example with **`into`**, here's a very simplified implementation of it that uses a transient for performance, returning a persistent data structure and thus exposing a purely functional interface although it uses mutation internally:

```
(defn my-into
  [to from]
  (persistent! (reduce conj! (transient to) from)))

(my-into [] (range 100))
;; => [0 1 2 ... 98 99]
```

## 5.3. Metadata

ClojureScript symbols, vars and persistent collections support attaching metadata to them. Metadata is a map with information about the entity it's attached to. The ClojureScript compiler uses metadata for several purposes such as type hints, and the metadata system can be used by tooling, library and application developers too.

There may not be many cases in day-to-day ClojureScript programming where you need metadata, but it is a nice language feature to have and know about; it may come in handy at some point. It makes things like runtime code introspection and documentation generation very easy. You'll see why throughout this section.

### 5.3.1. Vars

Let's define a var and see what metadata is attached to it by default. Note that this code is executed in a REPL, and thus the metadata of a var defined in a source file may vary. We'll use the **`meta`** function to retrieve the metadata of the given value:

```
(def answer-to-everything 42)
;; => 42

#'answer-to-everything
;; => #'cljs.user/answer-to-everyhing

(meta #'answer-to-everything)
;; => {:ns cljs.user,
;;     :name answer-to-everything,
;;     :file "NO_SOURCE_FILE",
;;     :source "answer-to-everything",
;;     :column 6,
;;     :end-column 26,
;;     :line 1,
;;     :end-line 1,
;;     :arglists (),
;;     :doc nil,
;;     :test nil}
```

Few things to note here. First of all, **`#'answer-to-everything`** gives us a reference to the **`Var`** that holds the value of the **`answer-to-everything`** symbol. We see that it includes information about the namespace (**`:ns`**) in which it was defined, its name, file (although, since it was defined at a REPL doesn't have a source file), source, position in the file where it was defined, argument list (which only makes sense for functions), documentation string and test function.

Let's take a look at a function var's metadata:

```
(defn add
  "A function that adds two numbers."
  [x y]
  (+ x y))
```

```
(meta #'add)
;; => {:ns cljs.user,
;;     :name add,
;;     :file "NO_SOURCE_FILE",
;;     :source "add",
;;     :column 7,
;;     :end-column 10,
;;     :line 1,
;;     :end-line 1,
;;     :arglists (quote ([x y])),
;;     :doc "A function that adds two numbers.",
;;     :test nil}
```

We see that the argument lists are stored in the `:arglists` field of the var's metadata and its documentation in the `:doc` field. We'll now define a test function to learn about what `:test` is used for:

```
(require '[cljs.test :as t])

(t/deftest i-pass
  (t/is true))

(meta #'i-pass)
;; => {:ns cljs.user,
;;     :name i-pass,
;;     :file "NO_SOURCE_FILE",
;;     :source "i-pass",
;;     :column 12,
;;     :end-column 18,
;;     :line 1,
;;     :end-line 1,
;;     :arglists (),
;;     :doc "A function that adds two numbers.",
;;     :test #<function (){ ... }>}
```

The `:test` attribute (truncated for brevity) in the `i-pass` var's metadata is a test function. This is used by the `cljs.test` library for discovering and running tests in the namespaces you tell it to.

### 5.3.2. Values

We learned that vars can have metadata and what kind of metadata is added to them for consumption by the compiler and the **`cljs.test`** testing library. Persistent collections can have metadata too, although they don't have any by default. We can use the **`with-meta`** function to derive an object with the same value and type with the given metadata attached. Let's see how:

```
(def map-without-metadata {:language "ClojureScript"})
;; => {:language "ClojureScript"}

(meta map-without-metadata)
;; => nil

(def map-with-metadata (with-meta map-without-metadata
                                  {:answer-to-everything 42}))
;; => {:language "ClojureScript"}

(meta map-with-metadata)
;; => {:answer-to-everything 42}

(= map-with-metadata
   map-without-metadata)
;; => true

(identical? map-with-metadata
            map-without-metadata)
;; => false
```

It shouldn't come as a surprise that metadata doesn't affect equality between two data structures since equality in ClojureScript is based on value. Another interesting thing is that **`with-meta`** creates another object of the same type and value as the given one and attaches the given metadata to it.

Another open question is what happens with metadata when deriving new values from a persistent data structure. Let's find out:

```
(def derived-map (assoc map-with-metadata :language "Clojure"))
;; => {:language "Clojure"}
```

```
(meta derived-map)
;; => {:answer-to-everything 42}
```

As you can see in the example above, metadata is preserved in derived versions of persistent data structures. There are some subtleties, though. As long as the functions that derive new data structures return collections with the same type, metadata will be preserved; this is not true if the types change due to the transformation. To ilustrate this point, let's see what happens when we derive a seq or a subvector from a vector:

```
(def v (with-meta [0 1 2 3] {:foo :bar}))
;; => [0 1 2 3]

(def sv (subvec v 0 2))
;; => [0 1]

(meta sv)
;; => nil

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

### 5.3.3. Syntax for metadata

The ClojureScript reader has syntactic support for metadata annotations, which can be written in different ways. We can prepend var definitions or collections with a caret (**^**) followed by a map for annotating it with the given metadata map:

```
(def ^{:doc "The answer to Life, Universe and Everything."} answer-to-
everything 42)
;; => 42

(meta #'answer-to-everything)
;; => {:ns cljs.user,
;;      :name answer-to-everything,
;;      :file "NO_SOURCE_FILE",
;;      :source "answer-to-everything",
;;      :column 6,
;;      :end-column 26,
```

```
;;      :line 1,
;;      :end-line 1,
;;      :arglists (),
;;      :doc "The answer to Life, Universe and Everything.",
;;      :test nil}

(def map-with-metadata ^{:answer-to-everything 42}
 {:language "ClojureScript"})
;; => {:language "ClojureScript"}

(meta map-with-metadata)
;; => {:answer-to-everything 42}
```

Notice how the metadata given in the **answer-to-everything** var definition is merged with the var metadata.

A very common use of metadata is to set certain keys to a **true** value. For example we may want to add to a var's metadata that the variable is dynamic or a constant. For such cases, we have a shorthand notation that uses a caret followed by a keyword. Here are some examples:

```
(def ^:dynamic *foo* 42)
;; => 42

(:dynamic (meta #'*foo*))
;; => true

(def ^:foo ^:bar answer 42)
;; => 42

(select-keys (meta #'answer) [:foo :bar])
;; => {:foo true, :bar true}
```

There is another shorthand notation for attaching metadata. If we use a caret followed by a symbol it will be added to the metadata map under the **:tag** key. Using tags such as **^boolean** gives the ClojureScript compiler hints about the type of expressions or function return types.

```
(defn ^boolean will-it-blend? [_] true)
```

```
;; => #<function ... >

(:tag (meta #'will-it-blend?))
;; => boolean

(not ^boolean (js/isNaN js/NaN))
;; => false
```

## 5.3.4. Functions for working with metadata

We've learned about **meta** and **with-meta** so far but ClojureScript offers a few functions for transforming metadata. There is **vary-meta** which is similar to **with-meta** in that it derives a new object with the same type and value as the original but it doesn't take the metadata to attach directly. Instead, it takes a function to apply to the metadata of the given object to transform it for deriving new metadata. This is how it works:

```
(def map-with-metadata ^{:foo 40} {:language "ClojureScript"})
;; => {:language "ClojureScript"}

(meta map-with-metadata)
;; => {:foo 40}

(def derived-map (vary-meta map-with-metadata update :foo + 2))
;; => {:language "ClojureScript"}

(meta derived-map)
;; => {:foo 42}
```

If instead we want to change the metadata of an existing var or value we can use **alter-meta!** for changing it by applying a function or **reset-meta!** for replacing it with another map:

```
(def map-with-metadata ^{:foo 40} {:language "ClojureScript"})
;; => {:language "ClojureScript"}

(meta map-with-metadata)
;; => {:foo 40}
```

```
(alter-meta! map-with-metadata update :foo + 2)
;; => {:foo 42}

(meta map-with-metadata)
;; => {:foo 42}

(reset-meta! map-with-metadata {:foo 40})
;; => {:foo 40}

(meta map-with-metadata)
;; => {:foo 40}
```

## 5.4. Macros

TBD

## 5.5. Core protocols

One of the greatest qualities of the core ClojureScript functions is that they are implemented around protocols. This makes them open to work on any type that we extend with such protocols, be it defined by us or a third party.

### 5.5.1. Functions

As we've learned in previous chapters not only functions can be invoked. Vectors are functions of their indexes, maps are functions of their keys and sets are functions of their values.

We can extend types to be callable as functions implementing the **`IFn`** protocol. A collection that doesn't support calling it as a function is the queue, let's implement **`IFn`** for the **`PersistentQueue`** type so we're able to call queues as functions of their indexes:

```
(extend-type PersistentQueue
  IFn
  (-invoke
    ([this idx]
```

```
      (nth this idx))))

(def q #queue[:a :b :c])
;; => #queue [:a :b :c]

(q 0)
;; => :a

(q 1)
;; => :b

(q 2)
;; => :c
```

## 5.5.2. Printing

For learning about some of the core protocols we'll define a **`Pair`** type, which will hold a pair of values.

```
(deftype Pair [fst snd])
```

If we want to customize how types are printed we can implement the **`IPrintWithWriter`** protocol. It defines a function called **`-pr-writer`** that receives the value to print, a writer object and options; this function uses the writer object's **`-write`** function to write the desired **`Pair`** string representation:

```
(extend-type Pair
  IPrintWithWriter
  (-pr-writer [p writer _]
    (-write writer (str "#<Pair " (.-fst p) "," (.-snd p) ">"))))
```

## 5.5.3. Sequences

In a previous section we learned about sequences, one of ClojureScript's main abstractions. Remember the **`first`** and **`rest`** functions for working with sequences? They are defined in the **`ISeq`** protocol, so we can extend types for responding to such functions:

```
(extend-type Pair
  ISeq
  (-first [p]
    (.-fst p))

  (-rest [p]
    (list (.-snd p))))

(def p (Pair. 1 2))
;; => #<Pair 1,2>

(first p)
;; => 1

(rest p)
;; => (2)
```

Another handy function for working with sequences is **`next`**. Although **`next`** works as long as the given argument is a sequence, we can implement it explicitly with the **`INext`** protocol:

```
(def p (Pair. 1 2))

(next p)
;; => (2)

(extend-type Pair
  INext
  (-next [p]
    (println "Our next")
    (list (.-snd p))))

(next p)
;; Our next
;; => (2)
```

Finally, we can make our own types seqable implementing the **`ISeqable`** protocol. This means we can pass them to **`seq`** for getting a sequence back.

ISeqable

```
(def p (Pair. 1 2))

(extend-type Pair
  ISeqable
  (-seq [p]
    (list (.-fst p) (.-snd p))))

(seq p)
;; => (1 2)
```

Now our **`Pair`** type works with the plethora of ClojureScript functions for working with sequences:

```
(def p (Pair. 1 2))
;; => #<Pair 1,2>

(map inc p)
;; => (2 3)

(filter odd? p)
;; => (1)

(reduce + p)
;; => 3
```

### 5.5.4. Collections

Collection functions are also defined in terms of protocols. For this section examples we will make the native JavaScript string act like a collection.

The most important function for working with collection is **`conj`**, defined in the **`ICollection`** protocol. Strings are the only type which makes sense to **`conj`** to a string, so the **`conj`** operation for strings will be simply a concatenation:

```
(extend-type string
  ICollection
  (-conj [this o]
    (str this o)))
```

```
(conj "foo" "bar")
;; => "foobar"

(conj "foo" "bar" "baz")
;; => "foobarbaz"
```

Another handy function for working with collections is **empty**, which is part of the **IEmptyableCollection** protocol. Let's implement it for the string type:

```
(extend-type string
  IEmptyableCollection
  (-empty [_]
    ""))

(empty "foo")
;; => ""
```

We used the **string** special symbol for extending the native JavaScript string. If you want to learn more about it check the section about extending JavaScript types.

## Collection traits

There are some qualities that not all collections have, such as being countable in constant time or being reversible. These traits are splitted into different protocols since not all of them make sense for every collection. For illustrating these protocols we'll use the **Pair** type we defined earlier.

For collections that can be counted in constant time using the **count** function we can implement the **ICounted** protocol. It should be easy to implement it for the **Pair** type:

```
(extend-type Pair
  ICounted
  (-count [_]
    2))

(def p (Pair. 1 2))
```

```
(count p)
;; => 2
```

Some collection types such as vectors and lists can be indexed by a number using the **`nth`** function. If our types are indexed we can implement the **`IIndexed`** protocol:

```
(extend-type Pair
  IIndexed
  (-nth
    ([p idx]
      (case idx
        0 (.-fst p)
        1 (.-snd p)
        (throw (js/Error. "Index out of bounds"))))
    ([p idx default]
      (case idx
        0 (.-fst p)
        1 (.-snd p)
        default))))

(nth p 0)
;; => 1

(nth p 1)
;; => 2

(nth p 2)
;; Error: Index out of bounds

(nth p 2 :default)
;; => :default
```

### 5.5.5. Associative

There are many data structures that are associative: they map keys to values. We've encountered a few of them already and we know many functions for working with them like **`get`**, **`assoc`** or **`dissoc`**. Let's explore the protocols that these functions build upon.

First of all, we need a way to look up keys on an associative data structure. The **`ILookup`** protocol defines a function for doing so, let's add the ability to look up keys in our **`Pair`** type since is an associative data structure that maps the indices 0 and 1 to values.

```
(extend-type Pair
  ILookup
  (-lookup
    ([p k]
      (-lookup p k nil))
    ([p k default]
      (case k
        0 (.-fst p)
        1 (.-snd p)
        default))))

(get p 0)
;; => 1

(get p 1)
;; => 2

(get p :foo)
;; => nil

(get p 2 :default)
;; => :default
```

For using **`assoc`** on a data structure it must implement the **`IAssociative`** protocol. For our **`Pair`** type only 0 and 1 will be allowed as keys for associating values. **`IAssociative`** also has a function for asking whether a key is present or not.

```
(extend-type Pair
  IAssociative
  (-contains-key? [_ k]
    (contains? #{0 1} k))

  (-assoc [p k v]
    (case k
      0 (Pair. v (.-snd p))
      1 (Pair. (.-fst p) v)
      (throw (js/Error. "Can only assoc to 0 and 1 keys")))))
```

```
(def p (Pair. 1 2))
;; => #<Pair 1,2>

(assoc p 0 2)
;; => #<Pair 2,2>

(assoc p 1 1)
;; => #<Pair 1,1>

(assoc p 0 0 1 1)
;; => #<Pair 0,1>

(assoc p 2 3)
;; Error: Can only assoc to 0 and 1 keys
```

The complementary function for **assoc** is **dissoc** and it's part of the **IMap** protocol. It doesn't make much sense for our **Pair** type but we'll implement it nonetheless. Dissociating 0 or 1 will mean putting a **nil** in such position and invalid keys will be ignored.

```
(extend-type Pair
  IMap
  (-dissoc [p k]
    (case k
      0 (Pair. nil (.-snd p))
      1 (Pair. (.-fst p) nil)
      p)))

(def p (Pair. 1 2))
;; => #<Pair 1,2>

(dissoc p 0)
;; => #<Pair ,2>

(dissoc p 1)
;; => #<Pair 1,>

(dissoc p 2)
;; => #<Pair 1,2>
```

```
(dissoc p 0 1)
;; => #<Pair ,>
```

Associative data structures are made of key and value pairs we can call entries. The **`key`** and **`val`** functions allow us to query the key and value of such entries and they are built upon the **`IMapEntry`** protocol. Let's see a few examples of **`key`** and **`val`** and how map entries can be used to build up maps:

```
(key [:foo :bar])
;; => :foo

(val [:foo :bar])
;; => :bar

(into {} [[:foo :bar] [:baz :xyz]])
;; => {:foo :bar, :baz :xyz}
```

Pairs can be map entries too, we treat their first element as the key and the second as the value:

```
(extend-type Pair
  IMapEntry
  (-key [p]
    (.-fst p))

  (-val [p]
    (.-snd p)))

(def p (Pair. 1 2))
;; => #<Pair 1,2>

(key p)
;; => 1

(val p)
;; => 2

(into {} [p])
;; => {1 2}
```

### 5.5.6. Comparison

For checking whether two or more values are equivalent with **=** we must implement the **IEquiv** protocol. Let's do it for our **Pair** type:

```
(def p  (Pair. 1 2))
(def p' (Pair. 1 2))
(def p'' (Pair. 1 2))

(= p p')
;; => false

(= p p' p'')
;; => false

(extend-type Pair
  IEquiv
  (-equiv [p other]
    (and (instance? Pair other)
         (= (.-fst p) (.-fst other))
         (= (.-snd p) (.-snd other)))))

(= p p')
;; => true

(= p p' p'')
;; => true
```

We can also make types comparable. The function **compare** takes two values and returns a negative number if the first is less than the second, 0 if both are equal and 1 if the first is greater than the second. For making our types comparable we must implement the **IComparable** protocol.

For pairs, comparison will mean checking if the two first values are equal. If they are, the result will be the comparison of the second values. If not, we will return the result of the first comparison:

```
(extend-type Pair
  IComparable
```

```
  (-compare [p other]
    (let [fc (compare (.-fst p) (.-fst other))]
      (if (zero? fc)
        (compare (.-snd p) (.-snd other))
        fc))))

(compare (Pair. 0 1) (Pair. 0 1))
;; => 0

(compare (Pair. 0 1) (Pair. 0 2))
;; => -1

(compare (Pair. 1 1) (Pair. 0 2))
;; => 1

(sort [(Pair. 1 1) (Pair. 0 2) (Pair. 0 1)])
;; => (#<Pair 0,1> #<Pair 0,2> #<Pair 1,1>)
```

## 5.5.7. Metadata

The **meta** and **with-meta** functions are also based upon two protocols: **IMeta** and **IWithMeta** respectively. We can make our own types capable of carrying metadata adding an extra field for holding the metadata and implementing both protocols.

Let’s implement a version of **Pair** that can have metadata:

```
(deftype Pair [fst snd meta]
  IMeta
  (-meta [p] meta)

  IWithMeta
  (-with-meta [p new-meta]
    (Pair. fst snd new-meta)))


(def p (Pair. 1 2 {:foo :bar}))
;; => #<Pair 1,2>

(meta p)
;; => {:foo :bar}
```

```
(def p' (with-meta p {:bar :baz}))
;; => #<Pair 1,2>

(meta p')
;; => {:bar :baz}
```

### 5.5.8. Interoperability with JavaScript

Since ClojureScript is hosted in a JavaScript VM we often need to convert ClojureScript data structures to JavaScript ones and viceversa. We also may want to make native JS types participate in an abstraction represented by a protocol.

#### Extending JavaScript types

When extending JavaScript objects instead of using JS globals like **`js/String`**, **`js/Date`** and such, special symbols are used. This is done for avoiding mutating global JS objects.

The symbols for extending JS types are: **`object`**, **`array`**, **`number`**, **`string`**, **`function`**, **`boolean`** and **`nil`** is used for the null object. The dispatch of the protocol to native objects uses Google Closure's goog.typeOf[1] function. There's a special **`default`** symbol that can be used for making a default implementation of a protocol for every type.

For illustrating the extension of JS types we are going to define a **`MaybeMutable`** protocol that'll have a **`mutable?`** predicate as its only function. Since in JavaScript mutability is the default we'll extend the default JS type returning true from **`mutable?`**:

```
(defprotocol MaybeMutable
  (mutable? [this] "Returns true if the value is mutable."))

(extend-type default
  MaybeMutable
  (mutable? [_] true))

;; object
(mutable? #js {})
```

[1] https://google.github.io/closure-library/api/namespace_goog.html#typeOf

```
;; => true

;; array
(mutable? #js [])
;; => true

;; string
(mutable? "")
;; => true

;; function
(mutable? (fn [x] x))
;; => true
```

Since fortunately not all JS object's values are mutable we can refine the implementation of **`MaybeMutable`** for returning **`false`** for strings and functions.

```
(extend-protocol MaybeMutable
  string
  (mutable? [_] false)

  function
  (mutable? [_] false))


;; object
(mutable? #js {})
;; => true

;; array
(mutable? #js [])
;; => true

;; string
(mutable? "")
;; => false

;; function
(mutable? (fn [x] x))
;; => false
```

There is no special symbol for JavaScript dates so we have to extend **`js/Date`** directly. The same applies to the rest of the types found in the global **`js`** namespace.

## Converting data

For converting values from ClojureScript types to JavaScript ones and viceversa we use the **`clj→js`** and **`js→clj`** functions, which are based in the **`IEncodeJS`** and **`IEncodeClojure`** protocols respectively.

For the examples we'll use the Set type introduced in ES6. Note that is not available in every JS runtime.

### From ClojureScript to JS

First of all we'll extend ClojureScript's set type for being able to convert it to JS. By default sets are converted to JavaScript arrays:

```
(clj->js #{1 2 3})
;; => #js [1 3 2]
```

Let's fix it, **`clj→js`** is supposed to convert values recursively so we'll make sure to convert all the set contents to JS and creating the set with the converted values:

```
(extend-type PersistentHashSet
  IEncodeJS
  (-clj->js [s]
    (js/Set. (into-array (map clj->js s)))))

(def s (clj->js #{1 2 3}))
(es6-iterator-seq (.values s))
;; => (1 3 2)

(instance? js/Set s)
;; => true

(.has s 1)
;; => true
(.has s 2)
;; => true
```

```
(.has s 3)
;; => true
(.has s 4)
;; => false
```

The **es6-iterator-seq** is an experimental function in ClojureScript core for obtaining a seq from an ES6 iterable.

### From JS to ClojureScript

Now it's time to extend the JS set to convert to ClojureScript. As with **clj→js**, **js→clj** recursively converts the value of the data structure:

```
(extend-type js/Set
  IEncodeClojure
  (-js->clj [s options]
    (into #{} (map js->clj (es6-iterator-seq (.values s))))))

(= #{1 2 3}
   (js->clj (clj->js #{1 2 3})))
;; => true

(= #{[1 2 3] [4 5] [6]}
   (js->clj (clj->js #{[1 2 3] [4 5] [6]})))
;; => true
```

Note that there is no one-to-one mapping between ClojureScript and JavaScript values. For example, ClojureScript keywords are converted to JavaScript strings when passed to **clj→js**.

## 5.5.9. Reductions

IReduce

IKVReduce

## 5.5.10. Asynchrony

IPending

IDeref

IDerefWithTimeout

### 5.5.11. State

IWatchable

IReset

ISwap

IVolatile

### 5.5.12. Mutation

IEditableCollection

ITransientCollection

ITransientAssociative

ITransientMap

ITransientVector

ITransientSet

## 5.6. The Reader

# Chapter 6. Appendix

## 6.1. Appendix A: Interactive development with Figwheel

### 6.1.1. Introduction

In this project, we will **not** do “Hello World”— that has been done to death. Instead, this project will be a web page that asks you for your age in years and tells you how many days that is, using an approximation of 365 days per year.

And for it, we will use the figwheel leiningen plugin. That plugin enables a fully interactive, repl based and autoreloading experience.

### 6.1.2. First steps

In this project, we will use the *figwheel* template to build the project structure. Let’s call the project **age** and create it by typing:

```
$ lein new figwheel age
Retrieving figwheel/lein-template/0.3.5/lein-template-0.3.5.pom from
 clojars
Retrieving figwheel/lein-template/0.3.5/lein-template-0.3.5.jar from
 clojars
Generating fresh 'lein new' figwheel project.
$ cd age # move into newly created project directory
```

The project has the following structure:

```
age
├── .gitignore
├── project.clj
├── README.md
├── resources
│   └── public
│       ├── css
│       │   └── style.css
│       └── index.html
```

```
└── src
    └── age
        └── core.cljs
```

The **project.clj** file contains information that Leiningen uses to download dependencies and build the project. For now, just trust that everything in that file is exactly as it should be.

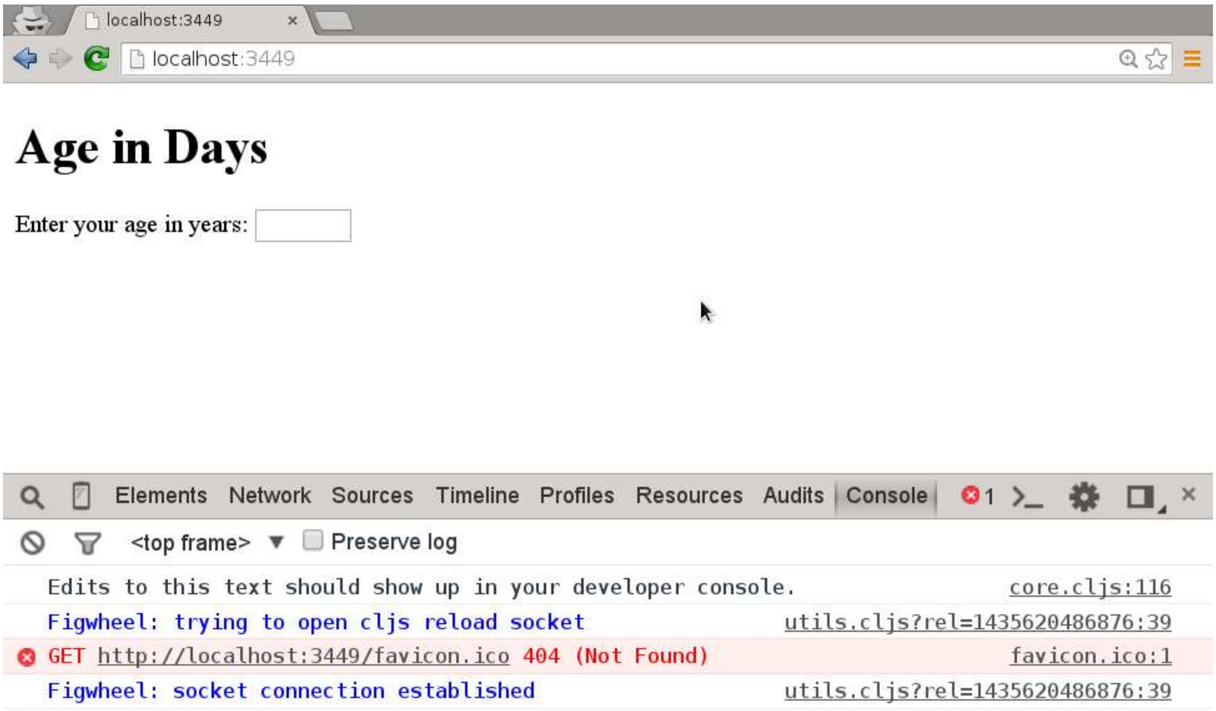
Open the **index.html** file and add the **<meta>** element to the head of the document, and modify the body as follows:

```
<!DOCTYPE html>
<html>
  <head>
    <link href="css/style.css" rel="stylesheet" type="text/css">
    <meta http-equiv="Content-Type" content="text/html;charset=utf-8" />
  </head>
  <body>
    <div id="app">
      <h1>Age in Days</h1>
      <p>Enter your age in years:
        <input type="text" size="5" id="years" /></p>
      <p id="feedback"></p>
    </div>
    <script src="js/compiled/age.js" type="text/javascript"></script>
  </body>
</html>
```

The **core.cljs** file is where all the action takes place. For now, leave it exactly as it is, and start the figwheel environment, which will load a large number of dependencies and start a server.

```
$ lein fighwheel
# much output
Prompt will show when figwheel connects to your application
```

If you are using Linux or Mac OS X, type the command as **rlwrap lein figwheel**. In your browser, go to URL **http://localhost:3449**, and you will see something like the following screenshot if you open up the web console.

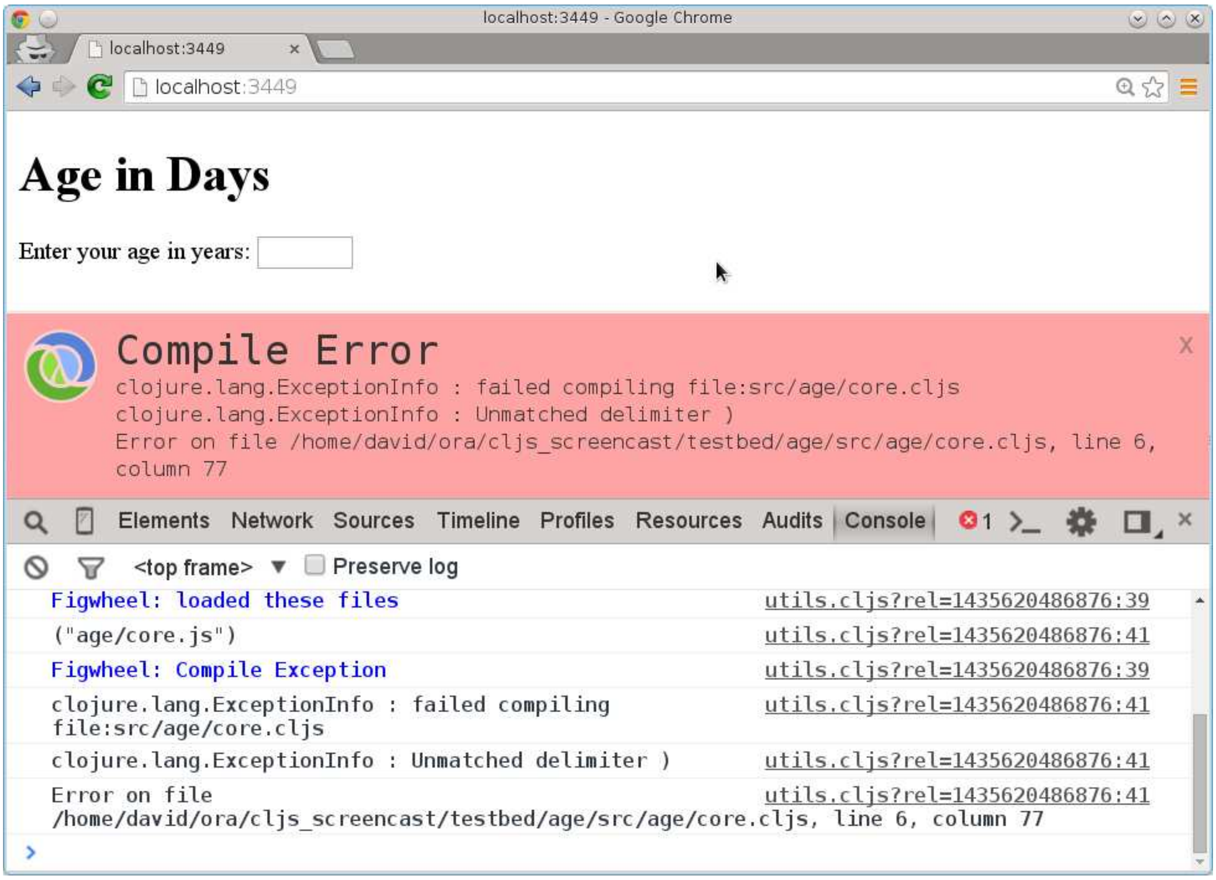



The terminal will then give you a REPL prompt:

```
$ rlwrap lein figwheel
To quit, type: :cljs/quit
cljs.user=>
```

For now, do what it says in the **`core.cljs`** file — change the **`(println…)`** and then save the file. When you do so, you will see the change reflected immediately in the browser.

Then make an error by adding an extra closing parenthesis to the **`println`**. When you save the file, will see a compile error in the browser window.




### 6.1.3. Interacting with JavaScript

In the REPL window, type the following to invoke JavaScript's **`window.alert()`** function:

```
(.alert js/window "It works!")
;; => nil
```

The general format for invoking a JavaScript function from ClojureScript is to give the function name (preceded by a dot), the object that "owns" the function, and any parameters to that function. You should see an alert appear in your browser winodw; when you dismiss the alert, the REPL will print **`nil`** and give you another prompt. You can also do it this way:

```
(js/alert "It works!")
;; => nil
```

However, the first version always works, so, for consistency, we will use that notation throughout. You can create objects by using their class name followed by a dot, and you can call methods on any JavaScript objects:

```
(def d (js/Date.))
;; => #inst "2015-06-30T01:38:21.764-00:00"

(.getFullYear d)
;; => 2015

(.toUpperCase "abc")
;; => "ABC"

(.getElementById js/document "years")
;; => #<[object HTMLInputElement]>
```

From that last example, you can see where we're going. To retrieve an object's property, use a `.-` before the property name. In the browser window, type a number into the input field (in the example, we typed `24`), then do this in the REPL.

```
;; for convenience define a variable
(def year-field (.getElementById js/document "years"))
;; => #<[object HTMLInputElement]>

(.-value year-field)
;; => "24"

(set! (.-value year-field) "25")
;; => "25"
```

This works, but it is little more than direct translation of JavaScript to ClojureScript. The next step is to add event handling to the button. Event handling is loaded with all sorts of cross-platform compatibility issues, so we'd like one step up from plain ClojureScript.

The solution is the Google Closure library. To use it, you have to modify the `:require` clause at the beginning of `core.cljs`:

```
(ns ^:figwheel-always age.core
```

```
  (:require [goog.dom :as dom]
            [goog.events :as events]))
```

Getting an element and setting its value is slightly easier. Do this in the REPL and see the results in the browser window.

```
(in-ns 'age.core)
(def y (dom/getElement "years"))
;; => #<[object HTMLInputElement]>

(set! (.-value y) "26")
;; => "26"

(dom/setTextContent (dom/getElement "feedback") "This works!")
```

To add an event, you define a function that takes a single argument (the event to be handled), and then tell the appropriate HTML element to listen for it. the **`events/listen`** function takes three arguments: the element to listen to, the event to listen for, and the function that will handle the event.

```
(defn testing [evt] (js/alert "Responding to click"))

(events/listen (dom/getElement "calculate") "click" testing)
;; => #<[object Object]>
```

After doing this, the browser should respond to a click on the button. If you would like to remove the listener, use **`unlisten`**.

```
(events/unlisten (dom/getElement "calculate") "click" testing)
;; => true
```

Now, put that all together in the **`core.cljs`** file as follows:

```
(ns ^:figwheel-always age.core
  (:require [goog.dom :as dom]
            [goog.events :as events]))

(enable-console-print!)
```

```
(defn calculate
  [event]
  (let [years (.parseInt js/window (.-value (dom/getElement "years")))
        days (* 365 years)]
    (dom/setTextContent (dom/getElement "feedback")
                        (str "That is " days " days old."))))

(defn on-js-reload [])

(events/listen (dom/getElement "calculate") "click" calculate)
```

## 6.2. Appendix B: CSP and core.async

CSP stands for Communicating Sequential Processes, which is a formalism for describing concurrent systems pioneered by C. A. R. Hoare in 1978. It is a concurrency model based on message passing and synchronization through channels. An in-depth look at the theoretical model behind CSP is out of the scope of this book, instead we'll focus on presenting the concurrency primitives that **`core.async`** offers.

**`core.async`** is not part of ClojureScript core but it's implemented as a library. Even though is not part of the core language it's widely used and many libraries build on top of its primitives so we think is worth covering in the book. It's also a good example of the syntactic abstractions that can be achieved transforming code with ClojureScript macros, so let's jump right in. You'll need to have **`core.async`** installed to run the examples presented in this section.

### 6.2.1. Channels

Channels are like conveyor belts, we can put and take a single value at a time from them. They can have multiple readers and writers and are the message-passing primitive of **`core.async`**. Let's create a channel do some operations on it:

```
(require '[cljs.core.async :refer [chan put! take!]])

(enable-console-print!)

(def ch (chan))
```

```
(take! ch #(println "Got a value:" %))
;; => nil

;; there is a now a pending take operation, let's put something on the
 channel

(put! ch 42)
;; Got a value: 42
;; => 42
```

In the above example we created a channel (**ch**) using the **chan** constructor. After that we performed a take operation on the channel, providing a callback that will be invoked when the take operation succeeds. After using **put!** to put a value on the channel the take operation completed and the **"Got a value: 42"** string was printed. Note that **put!** returned the value that was just put to the channel.

The **put!** function also accepts a callback like **take!** does but we didn't provide any in the last example. For puts the callback will be called whenever the value we provided has been taken. Puts and takes can happen in any order, let's do a few puts followed by takes to illustrate the point:

```
(require '[cljs.core.async :refer [chan put! take!]])

(def ch (chan))

(put! ch 42 #(println "Just put 42"))
;; => true
(put! ch 43 #(println "Just put 43"))
;; => true

(take! ch #(println "Got" %))
;; Got 42
;; Just put 42
;; => nil

(take! ch #(println "Got" %))
;; Got 43
;; Just put 43
;; => nil
```

You may be asking yourself why the **`put!`** operations return **`true`**. It signals that the put operation could be performed, even though the value hasn't yet been taken. Channels can be closed, which will cause the put operations to not succeed:

```
(require '[cljs.core.async :refer [chan put! close!]])

(def ch (chan))

(close! ch)
;; => nil

(put! ch 42)
;; => false
```

The above example was the simplest possible situation but what happens with pending operations when a channel is closed? Let's do a few takes and close the channel and see what happens:

```
(require '[cljs.core.async :refer [chan put! take! close!]])

(def ch (chan))

(take! ch #(println "Got value:" %))
;; => nil
(take! ch #(println "Got value:" %))
;; => nil

(close! ch)
;; Got value: nil
;; Got value: nil
;; => nil
```

We see that if the channel is closed all the **`take!`** operations receive a **`nil`** value. **`nil`** in channels is a sentinel value that signals to takers that the channel has been closed. Because of that, putting a **`nil`** value on a channel is not allowed:

```
(require '[cljs.core.async :refer [chan put!]])

(def ch (chan))
```

```
(put! ch nil)
;; Error: Assert failed: Can't put nil in on a channel
```

## Buffers

We've seen that pending take and put operations are enqueued in a channel but, what happens when there are many pending take or put operations? Let's find out by hammering a channel with many puts and takes:

```
(require '[cljs.core.async :refer [chan put! take!]])

(def ch (chan))

(dotimes [n 1025]
  (put! ch n))
;; Error: Assert failed: No more than 1024 pending puts are allowed on a
 single channel.

(def ch (chan))

(dotimes [n 1025]
  (take! ch #(println "Got" %)))
;; Error: Assert failed: No more than 1024 pending takes are allowed on a
 single channel.
```

As the example above shows there's a limit of pending puts or takes on a channel, it's currently 1024 but that is an implementation detail that may change. Note that there can't be both pending puts and pending takes on a channel since puts will immediately succeed if there are pending takes and viceversa.

Channels support buffering of put operations. If we create a channel with a buffer the put operations will succeed immediately if there's room in the buffer and be enqueued otherwise. Let's illustrate the point creating a channel with a buffer of one element. The **chan** constructors accepts a number as its first argument which will cause it to have a buffer with the given size:

```
(require '[cljs.core.async :refer [chan put! take!]])
```

```
(def ch (chan 1))

(put! ch 42 #(println "Put succeeded!"))
;; Put succeeded!
;; => true

(dotimes [n 1024]
  (put! ch n))
;; => nil

(put! ch 42)
;; Error: Assert failed: No more than 1024 pending puts are allowed on a
 single channel.
```

What happened in the example above? We created a channel with a buffer of size 1 and performed a put operation on it that succeeded immediately because the value was buffered. After that we did another 1024 puts to fill the pending put queue and, when trying to put one value more the channel complained about not being able to enqueue more puts.

Now that we know about how channels work and what are buffers used for let's explore the different buffers that **core.async** implements. Different buffers have different policies and it's interesting to know all of them to know when to use what. Channels are unbuffered by default.

### Fixed

The fixed size buffer is the one that is created when we give the **chan** constructor a number and it will have the size specified by the given number. It is the simplest possible buffer: when full, puts will be enqueued.

The **chan** constructor accepts either a number or a buffer as its first argument. The two channels created in the following example both use a fixed buffer of size 32:

```
(require '[cljs.core.async :refer [chan buffer]])

(def a-ch (chan 32))
```

```
(def another-ch (chan (buffer 32)))
```

## Dropping

The fixed buffer allows put operations to be enqueued. However, as we saw before, puts are still queued when the fixed buffer is full. If we wan't to discard the put operations that happen when the buffer is full we can use a dropping buffer.

Dropping buffers have a fixed size and, when they are full puts will complete but their value will be discarded. Let's illustrate the point with an example:

```
(require '[cljs.core.async :refer [chan dropping-buffer put! take!]])

(def ch (chan (dropping-buffer 2)))

(put! ch 40)
;; => true
(put! ch 41)
;; => true
(put! ch 42)
;; => true

(take! ch #(println "Got" %))
;; Got 40
;; => nil
(take! ch #(println "Got" %))
;; Got 41
;; => nil
(take! ch #(println "Got" %))
;; => nil
```

We performed three put operations and the three of them succeded but, since the dropping buffer of the channel has size 2, only the first two values were delivered to the takers. As you can observe the third take is enqueued since there is no value available, the third put's value (42) was discarded.

## Sliding

The sliding buffer has the opposite policy than the dropping buffer. When full puts will complete and the oldest value will be discarded in favor of the new one. The sliding buffer is useful when we are interested in processing the last puts only and we can afford discarding old values.

```
(require '[cljs.core.async :refer [chan sliding-buffer put! take!]])

(def ch (chan (sliding-buffer 2)))

(put! ch 40)
;; => true
(put! ch 41)
;; => true
(put! ch 42)
;; => true

(take! ch #(println "Got" %))
;; Got 41
;; => nil
(take! ch #(println "Got" %))
;; Got 42
;; => nil
(take! ch #(println "Got" %))
;; => nil
```

We performed three put operations and the three of them succeded but, since the sliding buffer of the channel has size 2, only the last two values were delivered to the takers. As you can observe the third take is enqueued since there is no value available since the first put's value was discarded.

## Transducers

As mentioned in the section about transducers, putting values in a channel can be thought as a transducible process. This means that we can create channels and hand them a transducer, giving us the ability to transform the input values before being put in the channel.

If we want to use a transducer with a channel we must supply a buffer since the reducing function that will be modified by the transducer will be the buffer's add function. A buffer's add function is a reducing function since it takes a buffer and an input and returns a buffer with such input incorporated.

```
(require '[cljs.core.async :refer [chan put! take!]])

(def ch (chan 1 (map inc)))

(put! ch 41)
;; => true

(take! ch #(println "Got" %))
;; Got 42
;; => nil
```

You may be wondering what happens to a channel when the reducing function returns a reduced value. It turns out that the notion of termination for channels is being closed, so channels will be closed when a reduced value is encountered:

```
(require '[cljs.core.async :refer [chan put! take!]])

(def ch (chan 1 (take 2)))

(take! ch #(println "Got" %))
;; => nil
(take! ch #(println "Got" %))
;; => nil
(take! ch #(println "Got" %))
;; => nil

(put! ch 41)
;; => true
(put! ch 42)
;; Got 41
;; => true
(put! ch 43)
;; Got 42
;; Got nil
;; => false
```

We used the **`take`** stateful transducer to allow maximum 2 puts into the channel. We then performed three take operations on the channel and we expect only two to receive a value. As you can see in the above example the third take got the sentinel **`nil`** value which indicates that the channel was closed. Also, the third put operation returned **`false`** indicating that it didn't take place.

## Handling exceptions

If adding a value to a buffer throws an exception **`core.async`** the operation will fail and the exception will be logged to the console. However, channel constructors accept a third argument: a function for handling exceptions.

When creating a channel with an exception handler it will be called with the exception whenever an exception occurs. If the handler returns **`nil`** the operation will fail silently and if it returns another value the add operation will be retried with such value.

```
(require '[cljs.core.async :refer [chan put! take!]])

(enable-console-print!)

(defn exception-xform
  [rfn]
  (fn [acc input]
    (throw (js/Error. "I fail!"))))

(defn handle-exception
  [ex]
  (println "Exception message:" (.-message ex))
  42)

(def ch (chan 1 exception-xform handle-exception))

(put! ch 0)
;; Exception message: I fail!
;; => true

(take! ch #(println "Got:" %))
;; Got: 42
;; => nil
```

## 6.2.2. Processes

We learned all about channels but there is still a missing piece in the puzzle: processes. Processes are pieces of logic that run independently and use channels for communication and coordination. Puts and takes inside a process will stop the process until the operation completes. Stopping a process doesn't block the only thread we have in the environments where ClojureScript runs. Instead, it will be resumed at a later time when the operation is waiting for has been performed.

Processes are launched using the **go** macro and puts and takes use the **<!** and **>!** placeholders. The **go** macro rewrites your code to use callbacks but inside **go** everything looks like synchronous code, which makes understanding it straightforward:

```
(require '[cljs.core.async :refer [chan <! >!]])
(require-macros '[cljs.core.async.macros :refer [go]])

(enable-console-print!)

(def ch (chan))

(go
  (println [:a] "Gonna take from channel")
  (println [:a] "Got" (<! ch)))

(go
  (println [:b] "Gonna put on channel")
  (>! ch 42)
  (println [:b] "Just put 42"))

;; [:a] Gonna take from channel
;; [:b] Gonna put on channel
;; [:b] Just put 42
;; [:a] Got 42
```

In the above example we are launching a process with **go** that takes a value from **ch** and prints it to the console. Since the value isn't immediately available it will park until it can resume. After that we launch another process that puts a value on the channel. Since there is a pending take the put operation succeeds and the value is delivered to the first process, then both processes terminate.

Both **go** blocks run independently and, even though they are executed asynchronously, they look like they synchronous code. The above go blocks are fairly simple but being able to write concurrent processes that coordinate via channels is a very powerful tool for implementing complex asynchronous workflows. Channels also offer a great decoupling of producers and consumers.

Processes can wait for an arbitrary amount of time too, there is a `timeout` function that return a channel that will be closed after the given amount of miliseconds. Combining a timeout channel with a take operation inside a go block gives us the ability to sleep:

```
(require '[cljs.core.async :refer [<! timeout]])
(require-macros '[cljs.core.async.macros :refer [go]])

(enable-console-print!)

(defn seconds
  []
  (.getSeconds (js/Date.)))

(println "Launching go block")

(go
  (println [:a] "Gonna take a nap" (seconds))
  (<! (timeout 1000))
  (println [:a] "I slept one second, bye!" (seconds)))

(println "Block launched")

;; Launching go block
;; Block launched
;; [:a] Gonna take a nap 9
;; [:a] I slept one second, bye! 10
```

As we can see in the messages printed, the process does nothing for one second when it blocks in the take operation of the timeout channel. The program continues and after one second the process resumes and terminates.

## Choice

Apart from putting and taking one value at a time inside a go block we can also make a non-deterministic choice on multiple channel operations using **alts!**. **alts!** is given a series of channel put or take operations (note that we can also try to put and take in a channel at the same time) and only performs one as soon as is ready; if multiple operations can be performed when calling **alts!** it will do a pseudo random choice by default.

We can easily try an operation on a channel and cancel it after a certain amount of time combining the **timeout** function and **alts!**. Let's see how:

```
(require '[cljs.core.async :refer [chan <! timeout alts!]])
(require-macros '[cljs.core.async.macros :refer [go]])

(enable-console-print!)

(def ch (chan))

(go
  (println [:a] "Gonna take a nap")
  (<! (timeout 1000))
  (println [:a] "I slept one second, trying to put a value on channel")
  (>! ch 42)
  (println [:a] "I'm done!"))

(go
  (println [:b] "Gonna try taking from channel")
  (let [cancel (timeout 300)
        [value ch] (alts! [ch cancel])]
    (if (= ch cancel)
      (println [:b] "Too slow, take from channel cancelled")
      (println [:b] "Got" value))))

;; [:a] Gonna take a nap
;; [:b] Gonna try taking from channel
;; [:b] Too slow, take from channel cancelled
;; [:a] I slept one second, trying to put a value on channel
```

In the example above we launched a go block that, after waiting for a second, puts a value in the **ch** channel. The other go block creates a **cancel** channel, which will be closed after 300 miliseconds. After that, it tries to read from both **ch** and **cancel** at the same time using **alts!**, which will succeed whenever it can take a value from either of those channels. Since **cancel** is closed after 300 miliseconds, **alts!** will succeed since takes from closed channel return the **nil** sentinel. Note that **alts!** returns a two-element vector with the returned value of the operation and the channel where it was performed. This is why we are able to detect whether the read operation was performed in the **cancel** channel or in **ch**. I suggest you copy this example and set the first process timeout to 100 miliseconds to see how the read operation on **ch** succeeds.

We've learned how to choose between read operations so let's look at how to express a conditional write operation in **alts!**. Since we need to provide the channel and a value to try to put on it, we'll use a two element vector with the channel and the value for representing write operations. Let's see an example:

```
(require '[cljs.core.async :refer [chan <! alts!]])
(require-macros '[cljs.core.async.macros :refer [go]])

(enable-console-print!)

(def a-ch (chan))
(def another-ch (chan))

(go
  (println [:a] "Take a value from `a-ch`")
  (println [:a] "Got" (<! a-ch))
  (println [:a] "I'm done!"))

(go
  (println [:b] "Take a value from `another-ch`")
  (println [:a] "Got" (<! another-ch))
  (println [:b] "I'm done!"))

(go
  (println [:c] "Gonna try putting in both channels simultaneously")
  (let [[value ch] (alts! [[a-ch 42]
                           [another-ch 99]])]
    (if (= ch a-ch)
```

```
      (println [:c] "Put a value in `a-ch`")
      (println [:c] "Put a value in `another-ch`"))))

;; [:a] Take a value from `a-ch`
;; [:b] Take a value from `another-ch`
;; [:c] Gonna try putting in both channels simultaneously
;; [:c] Put a value in `a-ch`
;; [:a] Got 42
;; [:a] I'm done!
```

When running the above example only the put operation on the **a-ch** channel has succeeded. Since both channels are ready to take a value when the **alts!** occurs you may get different results when running this code.

## Priority

**alts!** default is to make a non-deterministic choice whenever several operations are ready to be performed. We can instead give priority to the operations passing the **:priority** option to **alts!**. Whenever **:priority** is **true**, if more than one operation is ready they will be tried in order.

```
(require '[cljs.core.async :refer [chan >! alts!]])
(require-macros '[cljs.core.async.macros :refer [go]])

(enable-console-print!)

(def a-ch (chan))
(def another-ch (chan))

(go
  (println [:a] "Put a value on `a-ch`")
  (>! a-ch 42)
  (println [:a] "I'm done!"))

(go
  (println [:b] "Put a value on `another-ch`")
  (>! another-ch 99)
  (println [:b] "I'm done!"))

(go
```

```
  (println [:c] "Gonna try taking from both channels with priority")
  (let [[value ch] (alts! [a-ch another-ch] :priority true)]
    (if (= ch a-ch)
      (println [:c] "Got" value "from `a-ch`")
      (println [:c] "Got" value "from `another-ch`"))))

;; [:a] Put a value on `a-ch`
;; [:a] I'm done!
;; [:b] Put a value on `another-ch`
;; [:b] I'm done!
;; [:c] Gonna try taking from both channels with priority
;; [:c] Got 42 from `a-ch`
```

Since both **`a-ch`** and **`another-ch`** had a value to read when the **`alts!`** was executed and we set the **`:priority`** option to true, **`a-ch`** has preference. You can try deleting the **`:priority`** option and running the example multiple times to see that, without priority, **`alts!`** makes a non-deterministic choice.

## Defaults

Another interesting bit of **`alts!`** is that it can return immediately if no operation is ready and we provide a default value. We can conditionally do a choice on the operations if and only if any of them is ready, returning a default value if it's not.

```
(require '[cljs.core.async :refer [chan alts!]])
(require-macros '[cljs.core.async.macros :refer [go]])

(def a-ch (chan))
(def another-ch (chan))

(go
  (println [:a] "Gonna try taking from any of the channels without
 blocking")
  (let [[value ch] (alts! [a-ch another-ch] :default :not-ready)]
    (if (and (= value :not-ready)
             (= ch :default))
      (println [:a] "No operation is ready, aborting")
      (println [:a] "Got" value))))

;; [:a] Gonna try taking from any of the channels without blocking
```

```
;; [:a] No operation is ready, aborting
```

As you can see in the above example, if no operation is ready the value returned by **alts!** is the one we supplied after the **:default** key when calling it and the channel is the **:default** keyword itself.

### 6.2.3. Combinators

Now that we're acquainted with channels and processes it's time to explore some interesting combinators for working with channels that are present in **core.async**. This section includes a brief description of all of them together with a simple example of their usage.

#### pipe

**pipe** takes an input and output channels and pipes all the values put on the input channel to the output one. The output channel is closed whenever the source is closed unless we provide a **false** third argument:

```
(require '[cljs.core.async :refer [chan pipe put! <! close!]])
(require-macros '[cljs.core.async.macros :refer [go-loop]])

(def in (chan))
(def out (chan))

(pipe in out)

(go-loop [value (<! out)]
  (if (nil? value)
    (println [:a] "I'm done!")
    (do
      (println [:a] "Got" value)
      (println [:a] "Waiting for a value")
      (recur (<! out)))))

(put! in 0)
;; => true
(put! in 1)
;; => true
(close! in)
```

```
;; [:a] Got 0
;; [:a] Waiting for a value
;; [:a] Got 1
;; [:a] Waiting for a value
;; [:a] I'm done!
```

In the above example we used the **`go-loop`** macro for reading values recursively until the **`out`** channel is closed. Notice that when we close the **`in`** channel the **`out`** channel is closed too, making the **`go-loop`** terminate.

## pipeline-async

**`pipeline-async`** takes a number for controlling parallelism, an output channel, an asynchronous function and an input channel. The asynchronous function has two arguments: the value put in the input channel and a channel where it should put the result of its asynchronous operation, closing the result channel after finishing. The number controls the number of concurrent go blocks that will be used for calling the asynchronous function with the inputs.

The output channel will receive outputs in an order relative to the input channel, regardless the time each asynchronous function call takes to complete. It has an optional last parameter that controls whether the output channel will be closed when the input channel is closed, which defaults to **`true`**.

```
(require '[cljs.core.async :refer [chan pipeline-async put! <! close!]])
(require-macros '[cljs.core.async.macros :refer [go-loop]])

(def in (chan))
(def out (chan))
(def parallelism 3)

(defn wait-and-put [value ch]
  (let [wait (rand-int 1000)]
    (js/setTimeout (fn []
                     (println "Waiting" wait "miliseconds for value"
 value)
                     (put! ch wait)
                     (close! ch))
```

```
                  wait)))

(pipeline-async parallelism out wait-and-put in)

(go-loop [value (<! out)]
  (if (nil? value)
    (println [:a] "I'm done!")
    (do
      (println [:a] "Got" value)
      (println [:a] "Waiting for a value")
      (recur (<! out)))))

(put! in 1)
(put! in 2)
(put! in 3)
(close! in)

;; Waiting 164 miliseconds for value 3
;; Waiting 304 miliseconds for value 2
;; Waiting 908 miliseconds for value 1
;; [:a] Got 908
;; [:a] Waiting for a value
;; [:a] Got 304
;; [:a] Waiting for a value
;; [:a] Got 164
;; [:a] Waiting for a value
;; [:a] I'm done!
```

## pipeline

**pipeline** is similar to **pipeline-async** but instead of taking and asynchronous function it takes a transducer instead. The transducer will be applied independently to each input.

```
(require '[cljs.core.async :refer [chan pipeline put! <! close!]])
(require-macros '[cljs.core.async.macros :refer [go-loop]])

(def in (chan))
(def out (chan))
(def parallelism 3)
```

```
(pipeline parallelism out (map inc) in)

(go-loop [value (<! out)]
  (if (nil? value)
    (println [:a] "I'm done!")
    (do
      (println [:a] "Got" value)
      (println [:a] "Waiting for a value")
      (recur (<! out)))))

(put! in 1)
(put! in 2)
(put! in 3)
(close! in)

;; [:a] Got 2
;; [:a] Waiting for a value
;; [:a] Got 3
;; [:a] Waiting for a value
;; [:a] Got 4
;; [:a] Waiting for a value
;; [:a] I'm done!
```

## split

`split` takes a predicate and a channel and returns a vector with two channels, the first of which will receive the values for which the predicate is true, the second will receive those for which the predicate is false. We can optionally pass a buffer or number for the channels with the third (true channel) and fourth (false channel) arguments.

```
(require '[cljs.core.async :refer [chan split put! <! close!]])
(require-macros '[cljs.core.async.macros :refer [go-loop]])

(def in (chan))
(def chans (split even? in))
(def even-ch (first chans))
(def odd-ch (second chans))

(go-loop [value (<! even-ch)]
```

```
  (if (nil? value)
    (println [:evens] "I'm done!")
    (do
      (println [:evens] "Got" value)
      (println [:evens] "Waiting for a value")
      (recur (<! even-ch)))))

(go-loop [value (<! odd-ch)]
  (if (nil? value)
    (println [:odds] "I'm done!")
    (do
      (println [:odds] "Got" value)
      (println [:odds] "Waiting for a value")
      (recur (<! odd-ch)))))

(put! in 0)
(put! in 1)
(put! in 2)
(put! in 3)
(close! in)

;; [:evens] Got 0
;; [:evens] Waiting for a value
;; [:odds] Got 1
;; [:odds] Waiting for a value
;; [:odds] Got 3
;; [:odds] Waiting for a value
;; [:evens] Got 2
;; [:evens] Waiting for a value
;; [:evens] I'm done!
;; [:odds] I'm done!
```

## reduce

**reduce** takes a reducing function, initial value and an input channel. It returns a channel with the result of reducing over all the values put on the input channel before closing it using the given initial value as the seed.

```
(require '[cljs.core.async :as async :refer [chan put! <! close!]])
(require-macros '[cljs.core.async.macros :refer [go]])
```

```
(def in (chan))

(go
  (println "Result" (<! (async/reduce + (+) in))))

(put! in 0)
(put! in 1)
(put! in 2)
(put! in 3)
(close! in)

;; Result: 6
```

## onto-chan

**`onto-chan`** takes a channel and a collection and puts the contents of the collection into the channel. It closes the channel after finishing although it accepts a third argument for specifying if it should close it or not. Let's rewrite the **`reduce`** example using **`onto-chan`**:

```
(require '[cljs.core.async :as async :refer [chan put! <! close! onto-
chan]])
(require-macros '[cljs.core.async.macros :refer [go]])

(def in (chan))

(go
  (println "Result" (<! (async/reduce + (+) in))))

(onto-chan in [0 1 2 3])

;; Result: 6
```

## to-chan

**`to-chan`** takes a collection and returns a channel where it will put every value in the collection, closing the channel afterwards.

```
(require '[cljs.core.async :refer [chan put! <! close! to-chan]])
(require-macros '[cljs.core.async.macros :refer [go-loop]])

(def ch (to-chan (range 3)))

(go-loop [value (<! ch)]
  (if (nil? value)
    (println [:a] "I'm done!")
    (do
      (println [:a] "Got" value)
      (println [:a] "Waiting for a value")
      (recur (<! ch)))))

;; [:a] Got 0
;; [:a] Waiting for a value
;; [:a] Got 1
;; [:a] Waiting for a value
;; [:a] Got 2
;; [:a] Waiting for a value
;; [:a] I'm done!
```

## merge

**merge** takes a collection of input channels and returns a channel where it will put every value that is put on the input channels. The returned channel will be closed when all the input channels have been closed. The returned channel will be unbuffered by default but a number or buffer can be provided as the last argument.

```
(require '[cljs.core.async :refer [chan put! <! close! merge]])
(require-macros '[cljs.core.async.macros :refer [go-loop]])

(def in1 (chan))
(def in2 (chan))
(def in3 (chan))

(def out (merge [in1 in2 in3]))

(go-loop [value (<! out)]
  (if (nil? value)
```

```
    (println [:a] "I'm done!")
    (do
      (println [:a] "Got" value)
      (println [:a] "Waiting for a value")
      (recur (<! out)))))

(put! in1 1)
(close! in1)
(put! in2 2)
(close! in2)
(put! in3 3)
(close! in3)

;; [:a] Got 3
;; [:a] Waiting for a value
;; [:a] Got 2
;; [:a] Waiting for a value
;; [:a] Got 1
;; [:a] Waiting for a value
;; [:a] I'm done!
```

### 6.2.4. Higher-level abstractions

We've learned the about the low-level primitives of **`core.async`** and the combinators that it offers for working with channels. **`core.async`** also offers some useful, higher-level abstractions on top of channels that can serve as building blocks for more advanced functionality.

#### Mult

Whenever we have a channel whose values have to be broadcasted to many others, we can use **`mult`** for creating a multiple of the supplied channel. Once we have a mult, we can attach channels to it using **`tap`** and dettach them using **`untap`**. Mults also support removing all tapped channels at once with **`untap-all`**.

Every value put in the source channel of a mult is broadcasted to all the tapped channels, and all of them must accept it before the next item is broadcasted. For preventing slow takers from blocking the mult's values we must use buffering on the tapped channels judiciously.

Closed tapped channels are removed automatically from the mult. When putting a value in the source channels when there are still no taps such value will be dropped.

```
(require '[cljs.core.async :refer [chan put! <! close! timeout mult tap]])
(require-macros '[cljs.core.async.macros :refer [go-loop]])

;; Source channel and mult
(def in (chan))
(def m-in (mult in))

;; Sink channels
(def a-ch (chan))
(def another-ch (chan))

;; Taker for `a-ch`
(go-loop [value (<! a-ch)]
  (if (nil? value)
    (println [:a] "I'm done!")
    (do
      (println [:a] "Got" value)
      (recur (<! a-ch)))))

;; Taker for `another-ch`, which sleeps for 3 seconds between takes
(go-loop [value (<! another-ch)]
  (if (nil? value)
    (println [:b] "I'm done!")
    (do
      (println [:b] "Got" value)
      (println [:b] "Resting 3 seconds")
      (<! (timeout 3000))
      (recur (<! another-ch)))))

;; Tap the two channels to the mult
(tap m-in a-ch)
(tap m-in another-ch)

;; See how the values are delivered to `a-ch` and `another-ch`
(put! in 1)
(put! in 2)

;; [:a] Got 1
```

```
;; [:b] Got 2
;; [:b] Resting for 3 seconds
;; [:a] Got 2
;; [:b] Got 2
;; [:b] Resting for 3 seconds
```

## Pub-sub

After learning about mults you could imagine how to implement a pub-sub abstraction on top of **`mult`**, **`tap`** and **`untap`** but since it's a widely used communication mechanism **`core.async`** already implements this functionality. Instead of creating a mult from a source channel, we create a publication with **`pub`** giving it a channel and a function that will be used for extracting the topic of the messages.

We can subscribe to a publication with **`sub`**, giving it the publication we want to subscribe to, the topic we are interested in and a channel to put the messages that have the given topic. Note that we can subscribe a channel to multiple topics.

**`unsub`** can be given a publication, topic and channel for unsubscribing such channel from the topic. **`unsub-all`** can be given a publication and a topic to unsubscribe every channel from the given topic.

```
(require '[cljs.core.async :refer [chan put! <! close! pub sub]])
(require-macros '[cljs.core.async.macros :refer [go-loop]])

;; Source channel and publication
(def in (chan))
(def publication (pub in :action))

;; Sink channels
(def a-ch (chan))
(def another-ch (chan))

;; Channel with `:increment` action
(sub publication :increment a-ch)

(go-loop [value (<! a-ch)]
  (if (nil? value)
    (println [:a] "I'm done!")
```

```
    (do
      (println [:a] "Increment:" (inc (:value value)))
      (recur (<! a-ch)))))

;; Channel with `:double` action
(sub publication :double another-ch)

(go-loop [value (<! another-ch)]
  (if (nil? value)
    (println [:b] "I'm done!")
    (do
      (println [:b] "Double:" (* 2 (:value value)))
      (recur (<! another-ch)))))

;; See how values are delivered to `a-ch` and `another-ch` depending on
 their action
(put! in {:action :increment :value 98})
(put! in {:action :double :value 21})

;; [:a] Increment: 99
;; [:b] Double: 42
```

## Mixer

As we learned in the section about **`core.async`** combinators, we can use the **`merge`** function for combining multiple channels into one. When merging multiple channels, every value put in the input channels will end up in the merged channel. However, we may want more finer-grained control over which values put in the input channels end up in the output channel, that's where mixers come in handy.

**`core.async`** gives us the mixer abstraction, which we can use to combine multiple input channnels into an output channel. The interesting part of the mixer is that we can mute, pause and listen exclusively to certain input channels.

We can create a mixer given an output channel with **`mix`**. Once we have a mixer we can add input channels into the mix using **`admix`**, remove it using **`unmix`** or remove every input channel with **`unmix-all`**.

For controlling the state of the input channel we use the **`toggle`** function giving it the mixer and a map from channels to their states. Note that we can add channels to the mix using **toggle**, since the map will be merged with the current state of the mix. The state of a channel is a map which can have the keys **:mute**, **:pause** and **:solo** mapped to a boolean.

Let's see what muting, pausing and soloing channels means:

- A muted input channel means that, while still taking values from it, they won't be forwarded to the output channel. Thus, while a channel is muted, all the values put in it will be discarded.
- A paused input channel means that no values will be taken from it. This means that values put in the channel won't be forwarded to the output channel nor discarded.
- When soloing one or more channels the output channel will only receive the values put in soloed channels. By default non-soloed channels are muted but we can use **solo-mode** to decide between muting or pausing non-soloed channels.

That was a lot of information so let's see an example to improve our understanding. First of all, we'll set up a mixer with an **out** channel and add three input channels to the mix. After that, we'll be printing all the values received on the **out** channel to illustrate the control over input channels:

```
(require '[cljs.core.async :refer [chan put! <! close! mix admix unmix
 toggle solo-mode]])
(require-macros '[cljs.core.async.macros :refer [go-loop]])

;; Output channel and mixer
(def out (chan))
(def mixer (mix out))

;; Input channels
(def in-1 (chan))
(def in-2 (chan))
(def in-3 (chan))

(admix mixer in-1)
(admix mixer in-2)
(admix mixer in-3)

;; Let's listen to the `out` channel and print what we get from it
(go-loop [value (<! out)]
```

```
  (if (nil? value)
    (println [:a] "I'm done")
    (do
      (println [:a] "Got" value)
      (recur (<! out)))))
```

By default, every value put in the input channels will be put in the **`out`** channel:

```
(do
  (put! in-1 1)
  (put! in-2 2)
  (put! in-3 3))

;; [:a] Got 1
;; [:a] Got 2
;; [:a] Got 3
```

Let's pause the **`in-2`** channel, put a value in every input channel and resume **`in-2`**:

```
(toggle mixer {in-2 {:pause true}})
;; => true

(do
  (put! in-1 1)
  (put! in-2 2)
  (put! in-3 3))

;; [:a] Got 1
;; [:a] Got 3

(toggle mixer {in-2 {:pause false}})

;; [:a] Got 2
```

As you can see in the example above, the values put in the paused channels aren't discarded. For discarding values put in an input channel we have to mute it, let's see an example:

```
(toggle mixer {in-2 {:mute true}})
```

```
;; => true

(do
  (put! in-1 1)
  (put! in-2 2)  ;; `out` will never get this value since it's discarded
  (put! in-3 3))

;; [:a] Got 1
;; [:a] Got 3

(toggle mixer {in-2 {:mute false}})
```

We put a value (**2**) in the **in-2** channel and, since the channel was muted at the time, the value is discarded and never put into **out**. Let's look at the third state a channel can be inside a mixer: solo.

As we mentioned before, soloing channels of a mixer implies muting the rest of them by default:

```
(toggle mixer {in-1 {:solo true}
               in-2 {:solo true}})
;; => true

(do
  (put! in-1 1)
  (put! in-2 2)
  (put! in-3 3)) ;; `out` will never get this value since it's discarded

;; [:a] Got 1
;; [:a] Got 2

(toggle mixer {in-1 {:solo false}
               in-2 {:solo false}})
```

However, we can set the mode the non-soloed channels will be in while there are soloed channels. Let's set the default non-solo mode to pause instead of the default mute:

```
(solo-mode mixer :pause)
;; => true
(toggle mixer {in-1 {:solo true}
```

```
                in-2 {:solo true}})
;; => true

(do
  (put! in-1 1)
  (put! in-2 2)
  (put! in-3 3))

;; [:a] Got 1
;; [:a] Got 2

(toggle mixer {in-1 {:solo false}
               in-2 {:solo false}})

;; [:a] Got 3
```

## 6.3. Appendix C: Setting up a ClojureScript development environment

### 6.3.1. Cursive

TODO

### 6.3.2. Emacs

TODO

### 6.3.3. Vim

TODO

# Chapter 7. Acknowledgments

Special thanks to:

- **J David Eisenberg**: For the huge amount of time spend in fixing all kind of errors and writing entire sections of the book, as well as making very valuable suggestions.

And here is an inevitably incomplete list of MUCH-APPRECIATED CONTRIBUTORS — people who have submitted corrections, new ideas and generally made the *ClojureScript Unraveled* book much better:

- Anler Hernández Peral (@anler)
- Diego Sevilla Ruiz (@dsevilla)
- Eduardo Ferro Aldama (@eferro)
- Tyler Anderson (@Tyler-Anderson)
- Chris Ulrich (@chrisulrich)
- Jean Hadrien Chabran (@jhchabran)
- Tienson Qin (@tiensonqin)
- FungusHumungus (@FungusHumungus),
- Chris Charles (@ccharles)
- Jearvon Dharrie (@iamjarvo)
- Shaun LeBron (@shaunlebron)
- Wodin (@wodin)

# Chapter 8. Further Reading

Here is a list of more resources about ClojureScript.

- ClojureScript wiki[1]: a community-mantained wiki about ClojureScript.
- API Reference[2]: a community-maintained complete language api reference.
- ClojureScript Cheatsheet[3]: a comprehensive reference of the ClojureScript language.
- Études for ClojureScript[4]: a collection of exercises for learning ClojureScript.
- ClojureScript made easy[5]: a collection of short articles about solving common problems in ClojureScript.
- The Google Closure Library API reference[6]

1 https://github.com/clojure/clojurescript/wiki
2 https://github.com/cljsinfo/cljs-api-docs/blob/catalog/INDEX.md
3 http://cljs.info/cheatsheet/
4 http://catcode.com/etudes-for-clojurescript/toc01.html
5 http://clojurescriptmadeeasy.com/
6 http://google.github.io/closure-library/api/