What is the relationship between Common Lisp, Scheme, and Clojure?

Scheme and Clojure are in the lisp-1 family, which use the same namespace for functions and variables.
Common Lisp is in the lisp-2 family, which uses separate namespaces.

What's the easiest way to create a Clojure project?

With the leiningen build automation tool.

```
lein new MyProj
```

What is Clojure's function invocation syntax?

```
(funcname arg1 arg2 ...)
```

What is odd about Clojure operators?

- It uses the normal function prefix notation.
- `/` results in a `Ratio`.

How do you calculate remainders?

Using `mod`.

What is the advantage of doing math in prefix notation?

- It's easier to support higher-arity versions of the function.
- There's no ambiguity and no need for consideration of operator precedence.

How can you concatenate strings?

Using the equivalent of Java's `toString`.

```
(str obj1 obj2 ... objn)
```

Create a string without quotes.

Use \s:

```
user=> (str \h \i)
"hi"
```

Back slashes actually create characters, which are not strings in Clojure.

Show the Java class of an expression.

```
(class expr)
```

How are conditionals expressed?

Using an `if` function.
Its first argument is a Boolean expression, the second argument is the code to run if the Boolean expression is `true`. The third argument is optional and is the `else` code.

What evaluates to "true" for the purpose of `if` expressions?

Everything except `nil` and `false`.

`0` and `""` **do** evaluate to "true" in conditionals.

What conventionally separates lists and vectors?

What separates them in terms of implementation?

Lists are for code, while vectors are used to store data.

Vectors support fast random access.

How are non-function lists constructed?

With the `list` function or by quoting:

```
'("my" "favorite" "list")
```

Or with the `cons` function.

How do you get the head/tail of a list?

The last element?

```
first, rest, last
```

Access an arbitrary index in an list.

Combine two lists.

```
user=> (nth ["a", "b", "c"] 2)
"c"
```

Use `concat`.

What is the syntax of vectors, sets, and maps?

Vectors use `[]`, sets use `#{}`, maps use `{}`.

How do you assign to a variable?

Are re-assignments allowed?

```
(def VarName expr)
```

Yes, reassignments are allowed.

How can you access the size of collections?

Use `count`.

Show that vectors and sets are really functions.

```
user=> (["a" "b" "c"] 1)
"b"
user=> (#{"a" "b" "c"} "a")
"a"
```

What syntactic convenience does Clojure allow for for organizing members of lists, maps, etc?

Commas as whitespace.
With maps helps prevent mixing up keys/values, or putting an odd number of elements.

What are the two kinds of symbols in Clojure?

Keywords (beginning with `:`) stand for themselves, like atoms in other languages. Symbols refer to something else.

Retrieve a value from a map.

```
(:key map)
```

or

```
(map :key)
```

Both the map and the keyword are functions.

Add to a map.

`merge` and `merge-with` combine maps.

`assoc map newElement` adds to a map.

What is Clojure's function definition syntax?

```
(defn funcName [params] body)
```

Create and retrieve documentation.

Add a docstring after the function name and before the parameter vector. To retrieve use:

```
(doc funcName)
```

Where can destructuring be used?

How is it different from pattern matching?

It can be used in an argument list or in a `let` statement.

Unlike pattern matching, destructuring does not require you match the entire data structure being destructured.

What does `let` do?

```
(let [bindings* ] exprs*)
```

The expressions are evaluated in the lexical context of the bindings, sequentially applied.
The even numbered elements in `bindings` are bound to the odd symbols.
Unlike in a `def`, those bindings are not active outside the `let`.

Why are higher-order functions baked right into Clojure?

Because functions are really just lists.

How can you create anonymous functions?

Use `fn` instead of `defn` and don't put a name.

Or, put a `#` before the list containing the body of the function with `%` being bound to each argument.

How can you apply a function that is the result of an expression?

```
(apply func arglist)
```

How does Clojure work around the JVM's lack of tail recursion optimization?

Using `loop`/`recur`.

The `loop` function takes a vector whose odd number elements are variables and even number elements are initial variables to those arguments. Its second argument is the body of the function which can use `recur` to go back to `loop`.

Translate Scala's `forall`.

Translate Scala's `exists`.

Translate Java's `null` check.

`(every? func list)`

`(some func list)`

`not-every` and `not-any` are the opposite.

`(nil? list)`

How are list comprehensions done?

```
(for [el1 list1 ... el2 list2 :when filter1] body)
```

What is idiomatic Clojure for Java's `isX` functions?

What is idiomatic Clojure's capitalization system for functions.

`x?`

`word1-word2` type function names.

Scala's `foldLeft` goes by what in other languages?

- Erlang - `foldl`
- Ruby - `inject`
- Clojure - `reduce`
- Haskell - `foldl`

How do you create ranges?

```
user=> (range -5 5 2)
(-5 -3 -1 1 3)
```

Create an infinite sequence.

```
(repeat "to infinity and beyond!")

(repeat (cycle ["one" "more" "time"]))

(iterate inc 3)
```

Infinite sequences are typically combined with what?

Calls to `take` or `drop`, to extract sub-sequences.

What is Clojure's alternative to building things "inside out"?

Using the left-to-right operator `->>`. It applies each function to the result of the *previos* one.