What are Clojure's basic numerical types?

Integers, floating point numbers, ratios, and characters.

What are "keywords" in Clojure terminology?

Atoms, as in Prolog or Erlang, not reserved words.

Anonymous functions are also called ...

... lambdas.

What is Clojure's fundamental collection type?

What methods define it?

Sequences are the abstraction above lists, vectors, maps, sets, etc.

They share the methods `first`, `rest`, and `cons`.

What's the problem with `if`?

What's the alternative?

The nesting of many `if`s is difficult to read.

The special form `cond` allows as many arguments as needed for *if ... then ... elseif ... then ... elseif ...*
`cond` provides `:else` to catch everything as the final branch.

Sequences are roughly equivalent to what in Java?

# Iterators

What do predicates return in idiomatic Clojure?

Instead of returning the atoms `false`/`true`, they take advantage of the fact that anything other than `false`/`nil` is "true" and return a more meaningful value.

How can you create a list without invoking it as a function?

By quoting:

```
'(a b c)

(quote (a b c))
```

How are special forms different from functions?

What's the catch?

Special forms get arguments unevaluated, controlling if/when to evaluate them.

However, special forms are *not* first-class values.

Give two methods of division.

`/` which returns a `Ratio`.

`quot` performs truncating integer division.

What's the difference between `rem` and `mod`?

They differ in their handling of signs. If the first and second arguments have different signs, the result of `mod` will have the same sign as the second argument, while the result of `rem` will have the same sign as the first argument.

Describe negation.

`not` and `not=` are provided.

What's the difference between $=$ and $==$?

== only compares arguments that can be case to `java.lang.Number`.

= compares arguments in a type-independent manner. For example, vectors can be equal to lists according to =.

Unlike other Lisps, the order of functions ...

... matters. Forward-references are not allowed.

Test object identity.

identical

Test for collection type.

`coll?`, `seq?`, `vector?`, `list?`, `map?`, `set?`

Test a collection for membership.

For non-reptition.

`contains?`, `distinct?`

In Clojure the standard streams have ...

... special reassignable names.

`*in*`, `*out*`, `*err*`

Get back recent repl values.

`*1`, `*2`, `*3` are the first, second, and third most recent values.
`*e` is the most recent exception.

What is the comment syntax?

`;` comments out the rest of the line.

In place of classes, Clojure has ...

What is their syntax?

... structs

Define with:

```
(defstruct name :field1 :field2 ...)
```

Instantiate with:

```
(struct name val1 val2 ...)
```

Access fields with:

```
(:fieldName structName)
```

What is a lazy data structure?

What's an easy way to make them in Haskell?

A data structure whose parts don't exist until they are accessed.

`iterate` takes a function `f` and a starting value `n` and produces a lazy infinite series:

```
(n, f(n), f(f(n)), f(f(f(n))), ...)
```

What does the `&` mean in a function signature?

It comes before a vararg, which is available as a list in the body.

Describe `do`.

```
(do exprs*)
```

Evaluates the expressions in order and returns the value of the last.

Describe list comprehension syntax.

```
(for seq-exprs body-expr)
```

Takes a vector of one or more binding-form/collection-expr pairs, each followed by zero or more modifiers, and yields a lazy sequence of evaluations of expr.

Supported modifiers are: `:let [binding-form expr ...]`, `:while test`, `:when test`.

How do you partially apply functions?

Use `partial`, followed by a function and fewer than the normal number of arguments.

```
user=> (def equals5 (partial = 5))
#'user/equals5
user=> (equals5 5)
true
```