What is `()`?

Both the type and sole value of the empty tuple.

What are Haskell's capitalization rules?

Types start with an uppercase character, everything else with a lowercase letter.

What is the key benefit of type variables over generics?

Type variables stay maximally general (polymorphic), being restricted only by the ways the type is used. Generics force you to anticipate the maximum generality yourself when declaring the generic type's bounds.

From an object-oriented point of view, a typeclass can be thought of as ...

... an interface. It creates a contract that defines the behaviors of types of that class.

From Haskell's point of view an operator is different from a function only in that ...

... its name consists of only special characters and is infix by default.

What does => do in a type signature?

It indicates a class constraint. The left hand side indicates class membership/s (comma separated) of the type variables used on the right hand side.

What is `Eq`?

What are some prominent non-`Eq`s?

The type class for types that can be tested for equality.

`IO` and functions are not members.

Describe Haskell's equivalent of Java's `Comparable`.

The typeclass `Ord` defines a function `compare` which returns an `Ordering`, meaning either `GT`, `EQ`, or `LT`.

What is the `Show` typeclass?

Its members can be presented as strings using the `show` function.

How do you go back from `show`'s string representation to the object itself?

What's the catch?

The `Read` typeclass defines `read` which does just that.

However, in order to know what type is desired, the result of `read` must be used in an expression or given a type annotation.

What is a type annotation?

Give its syntax.

Haskell's term for a manifest type. It's not an annotation on a type in the sense of Scala's `@specialized`.

Haskell type annotations use the `::` followed by the type.

Members of `Enum` can do what?

What about `Bounded`?

They can be enumerated.

They have an upper and a lower bound: `maxBound` and `minBound`.

Give two examples of polymorphic constants.

```
ghci> :t 0
0 :: (Num t) => t
ghci> :t maxBound
maxBound :: (Bounded a) => a
```

Make integral and floating point types play nice together.

What is that function's signature?

```
ghci> :t fromIntegral
fromIntegral :: (Integral a, Num b) => a -> b
```

How can repeated computations be avoided in a series of guards?

With a `where` clause following the guards, containing variable assignments, one per line, aligned.

What additional kind of assignment can be made with `where`/`let`, pattern matching aside?

Named functions can be created, using the normal syntax.

Where are helper functions typically defined, when scope-limiting is desired?

In a `where` clause. Sometimes this gets nested into one helper helping another, and so on.

Other than order, what's the difference between `let/in` and `where`?

- `let` is an expression, while `where` is just a syntactic construct.
- `where` doesn't have such a tight scope, so its bindings are active across multiple guards.

How can multiple `let` bindings be put on one line?

By separating the assignments with semicolons.

In addition to guards and generators, what can go in list comprehensions?

`let` expressions, being far more powerful than Scala's midstream definitions.

Pattern matching on function parameters is just syntactic sugar for ...

What is the full syntax?

... case expressions.

```
case expr of pat1 -> res1
              ...
             patN -> resN
```