What is the basic syntax of the `data` keyword?

```
data TypeName = ValueConstructorA FieldA1 ... FieldAN | ValueConstructorB ...
```

What are value constructors, really?

What aren't they?

Functions that ultimately return a value of a data type. So the "fields" of a value constructor are really parameters.

They are **not** types.

Literals can be thought of as what?

Value constructors with no parameters.

How can you make your typeclass inherit another one?

Add `deriving (OtherClass)` to the end of the `data` declaration.

When there's only one value constructor for a type, it's common to do what?

Use the same name for the data type and the value constructor.

How do you export types from a module?

```
module MyModule
( DataConstructor1(..)
, ...
, DataConstructorN
) where ...
```

The `..` indicates all value constructors of the type constructor should be exported.
Otherwise you can comma separate the ones you want to export, or leave off the parens altogether to not export any value constructors.

What happens if you don't export any value constructors?

Give an example of a standard library type like this.

Instances of the type can be created only through auxiliary functions.

`Data.Map` can be constructed only through auxiliary functions like `fromList`.

What is record syntax?

Give its syntax.

A way of creating value constructors with automatic meaningfully named accessors.

```
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     , height :: Float
                     , phoneNumber :: String
                     , flavor :: String
                     } deriving (Show)
```

What use-site convenience does record syntax provide?

Named parameters. Users can invoke the value constructor using curlies and comma-separated named values.

Define `Maybe`.

Like Scala's `Option`:

```
data Maybe a = Nothing | Just a
```

When should you use type parameters?

When should you not?

Use them when your type is a container of some kind. Otherwise don't, since your functions on the type will make strong assumptions about the types filled in.

Why is there a strong convention against putting typeclass constraints in data declarations?

Constraints should follow from functions that assume the constraint.
If you put the constraint on the data declaration you will have to put the constraint on **all** functions, even when the constraint is irrelevant to that particular function.

What's the catch when deriving `Eq`?

All the field types must also be of the `Eq` typeclass.

What is the default interpretation of deriving `Ord`?

Ordering the value constructors least to greatest, in the order defined.

What is a value constructor with no parameters called?

nullary

What syntactic convenience to `Enums` have?

They can be used in list ranges.

Describe type synonyms?

Haskell's term for Scala's type aliases.
The syntax is the same as Scala's:

```
type Name = Type
```

What abilities do type synonyms have over Scala's type aliases?

Haskell's can be parameterized:

```
type AssocList k v = [(k,v)]
```

Haskell's can be "partially applied" using either of these notations:

```
type AssocListInt = AssocList Int
type AssocListInt v = AssocList Int v
```

What is a concrete type?

A type that is fully applied, with no unbound type parameters.

Describe `Either`.

What is it used for.

Approximately:

```
data Either a b = Left a | Right b deriving (Eq, Ord, Read, Show)
```

Errors use `Left`, successful result uses `Right`.

Its used for functions that can fail in multiple ways. `Maybe`'s `Nothing` would not inform as to which way it failed.

Demonstarte recursive types by defining `List` with standards syntax.

Do the same with record syntax.

What is a fixity declaration?

Give its syntax.

It indicates which way an operator associates and how tightly.

```
fixity num op
```

Where `fixity` is one of `infix`, `infixr`, `infixl`, and `num` is the tightness of binding, and `op` is the operator in question. Larger numbers indicate tighter bindings.

Give the `data` definition of a tree.

```
data Tree a = EmtpyTree | Node a (Tree a) (Tree a) deriving (Show, Read, Eq)
```

Give the basic typeclass syntax.

```
class ClassName typevar1 ... typevarn where
    funcDefOrDeclaration1
    ...
    funcDefOrDeclarationN
```

Function declarations are simply type signatures like `ghci` would put out in response to `:t`.

Manually create an instance of `Eq`.

```
instance Eq TrafficLight where
    Red == Red = True
    Green == Green = True
    Yellow == Yellow = True
    _ == _ = False
```

How can a typeclass implement all its functions yet require that instances implement some of them?

The typeclass might implement functions in terms of one another, requiring a minimal complete definition to define one to bootstrap the others.

What is *subclassing* in Haskell?

Given an example.

Adding a class constraint to a `class` declaration.

```
class (Eq a) => Num a where
    ...
```

How can you find information about names that aren't expressions?

`:info Name` in `ghci`.

What's the `Functor` typeclass?

Give a partial implementation.

It's for things that can be mapped over, like lists.

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

What is a *kind*?

How can the be found using `ghci`?

More or less, the type of a type.

```
:k Type
```