This package pertains to the type

```
Stream f m r
```

In what follows, whatever fills the `f` position is called *the streamed functor*, or *the form of the steps* or simply *the functor*. Whatever fills the `m` parameter is *the monad*, or *the monad of effects* or just the *effect* or *action* type. Finally whatever fills the `r` position is the *return* or *exit* type.

A value of type `Stream f m r` will be a succession of steps – potentially empty, and potentially infinite – with a structure determined by `f`, and arising from actions in the monad `m`, and returning a value of type `r`. We might depict the possibilities for individual such 'streams' like this:

```
...
m f (m f (m f (m f (m r))))  -- four steps remain
m f (m f (m f (m r)))        -- three steps remain
m f (m f (m r))              -- two steps remain
m f (m r)                    -- one step remains
m r                          -- done
```

`Streaming.Prelude`, and this tutorial, are focussed on a particular reading of the functor position:

```
Stream (Of a) m r
```

This is the *stream of individual Haskell values derived from actions in some monad `m` and returning a value of type r.* It is equivalent to `Producer a m r` in `pipes`, `ConduitM () o m r` in `conduit` and `Generator a r` in `io-streams`.

`Stream (Of a)` is a monad transformer, and, where `m` is a monad, `Stream (Of a) m` is of course always a functor and a monad. *The fundamental action in this monad is expressed by the `yield` statement, discussed below.*

The "streamed functor" here, `Of a`, is almost as minimal as can be - the left-strict pair:

```
data Of a r = !a :> r
```

which we use to link a `yield`-ed item to the rest of the stream. We only prefer this to the standard Haskell pair

```
data (,) a b = (a,b)
```

which is lazy in both positions, because `ghc` has proven better able to optimize it in our use case.

`Stream (Of a) m r` (like the isomorphic types from the standard streaming-io libraries) is an effectful variant of the Haskell type

```
([a],r)
```

Streams of this basic form might be depicted thus:

```
...
m (a :> m (a :> m (a :> m (a :> r)))) -- four steps remain
m (a :> m (a :> m (a :> m r)))        -- three steps remain
m (a :> m (a :> m r))                 -- two steps remain
m (a :> m r)                          -- one step remains
m r                                   -- done
```

When we run the outermost monadic action, we are either done (the last case), or we get a pair `a :> rest`, where `rest` is the rest of the stream. Given a stream, we can do this manually with `next`, which is discussed below. It is identical in content with `Pipes.next`:

```
>>> :t next
next :: Monad m => Stream (Of a) m r -> m (Either r (a, Stream (Of a) m r))
>>> :t Pipes.next
Pipes.next :: Monad m => Producer a m r -> m (Either r (a, Producer a m r))
```

If we read `m` as `Identity` and `r` as `()`, then it is of course isomorphic to `[a]` (but strict in the leaves). The `Show` instance applies at this type:

```
>>> each [1,2,3] :: Stream (Of Int) Identity ()
Step (1 :> Step (2 :> Step (3 :> Return ())))
```

The `Show` instance is derived, and thus exhibits the hidden constructors. Because `each` is a pure operation, we see no trace of any monadic action, just the functorial steps. We can perceive the more general case, in which functor steps are interleaved with monadic actions that may determine their content, we apply `S.mapM return`:

```
>>> S.mapM return $ each [1,2,3] :: Stream (Of Int) Identity ()
Effect (Identity (Step (1 :> Effect (Identity (Step (2 :> Effect (Identity (Step (3 :> Return ())
```

Here, all of the effects are predetermined, so we can again perceive the we just have a somewhat decorated variant of `1 : 2 : 3 : []`. The constructors are hidden in this library precisely in order to preserve the equivalence of the two streams above, i.e. so that things like `S.mapM return = id` hold. The second might be said to be the 'canonical' representative of quotient we operate with. The closest we come to pattern matching on the constructors, is by application of `next`. If you do not import `Streaming.Internal`, then the `Show` instance for `Stream (Of a) Identity r` is the only way you can observe it - and even then, all you can do is literally observe it, you cannot act on it. It would be possible to dispense with this apparatus of hiding, as in `Control.Monad.Trans.FreeT`, but the type would no longer be practical; the ordinary workings of `ghc` would have no room to simplify and optimize complex embedded loops.

But, to return to our leading types - `Stream f m r` and its specialization `Stream (Of a) m r` - it is essential to the whole construction that we permit a return type, `r`. Suppose we forbade it, and replaced the `Return ()` that we see in:

```
>>> each [1,2,3] :: Stream (Of Int) Identity ()
Step (1 :> Step (2 :> Step (3 :> Return ())))
```

with say `Nil` (i.e. the `[]` of Haskell lists). The we would have the familiar `ListT m a` type. This is a perfectly legitimate type, when 'done right', but *it cannot support any but the most primitive list operations.* `Stream f m r` could be expressed in various ways, but something with its internal complexity is essential to the reconstitution of the API of the Haskell `Prelude` and `Data.List`.

For example, we want to be able to express the streaming division of a stream. For example, we want to be able to break after the first two items streamed, whatever they might be, and return 'rest of the stream', proposing to handle it separately. That is, we want to be able to define `splitAt`:

```
>>> S.splitAt 2 $ each [1..4] :: Stream (Of Int) Identity (Stream (Of Int) Identity ())
Step (1 :> Step (2 :> Return (Step (3 :> Step (4 :> Return ())))))
```

This is impossible in `io-streams`, `machines` and `list-t`; it is possible in `conduit`, but is not supported, since it cannot be made to fit with the rest of the framework. `list-t` at least sees the necessity of some such operation, but must type it thus (specializing):

```
splitAt :: Monad m => Int -> ListT m a -> m ([a],ListT m a)
```

That is, in order to contemplate acting on the first n members in one way, and the rest in another way, I must break streaming and accumulate a Haskell list.

And we want to stream our streams - for example by dividing a given stream into segments - and operate on the substreams separately.

```
>>> chunksOf 2 $ each [1,2,3,4,5,6] :: Stream (Stream (Of Int) Identity) Identity ()
Step (Step (1 :> Step (2 :> Return (Step (Step (3 :> Step (4 :> Return (Step (Step (5 :> Step (6 :>
```

In principle, `f` might be any functor and `m` any monad. But we are interested in a quite particular range of functors - basically, things we can envisage making a left fold over - and on readings of `m` as `IO`, or as meeting some `MonadIO` constraint. *`Stream f m r` preserves these properties, and can thus reasonably be put in either position.* Inevitably we will also find constant use for types like

```
Stream (Stream (Of a) m) m r
```

and

```
Stream (Of a) (Stream (Of a) m) r
```

Because it massively overlaps with the `Prelude`, `Streaming.Prelude` must be imported qualified. The `Streaming` module, by contrast, is designed to be imported without qualification. Several important operations - in particular `each`, `next`, `yield` - are put in the `Streaming.Prelude` because they overlap with the `Pipes` module and are in all cases semantically very close to their `Pipes` meaning. In the present tutorial, `Pipes` is not at issue; the examples will thus presuppose the following imports:

```
import Streaming
import qualified Streaming.Prelude as S
import Streaming.Prelude (each, next, yield)
```

Occasionally we will see others, like

```
import qualified Control.Foldl as L -- cabal install foldl
```

Introducing a stream, easy cases

`yield`

The simplest and most rudimentary way to construct a stream is with the `yield` statement:

```
>>> :t yield
yield :: Monad m => a -> Stream (Of a) m ()

>>> S.print $ yield True
True
```

`yield` statements can be sequenced with `>>`, or with `do` notation:

```
>>> S.print $ yield True >> yield False
True
False

>>> S.stdoutLn $ do {yield "hello"; yield "world"}
hello
world
```

`yield` statements are the basic building block of any hand-written definition `Stream (Of a) m r`, and will be discussed further below.

each

While `yield` makes a singleton stream, `each` streams any pure, `Foldable` container of Haskell values into a nominally effectful stream:

```
>>> :t each [1..3::Int]
each [1..3::Int] :: Monad m => Stream (Of Int) m ()

>>> S.print $ each [1..3]
1
2
3
```

Even simple combinators can give us a genuinely effectful stream:

```
>>> :t S.replicateM 2 getLine
S.replicateM 2 getLine :: Stream (Of String) IO ()

>>> S.print $ S.replicateM 2 getLine
hello<Enter>
"hello"
world<Enter>
"world"
```

The simple textual IO operations, like `stdinLn`, `readFile` act line by line

If you are not familiar with it, `runResourceT` here amounts basically `close_handles`; it makes it possible to define a semi-sensible `readFile` and `writeFile` without explicit use of handles.

```
>>>

>>> runResourceT $ S.print $ S.readFile "hello.txt"
"hello"
"world"
```

Note that `readFile`, `stdinLn` and similar functions in `Streaming.Prelude` are line-based, and hold or act on regular Haskell `String`s.

Eliminating streams

These are ways of introducing, 'constructing' or 'unfolding' a stream; what are some ways of eliminating them? We have already been using `S.print`, which just prints all the elements, and keeps the final return value of the stream. Similarly `S.stdoutLn` eliminates a stream of strings by writing them on separate lines.

The easiest way 'reduce' a stream is with standard folds like `S.sum`, `S.product`, `S.toList` and the like. `S.print` is a minimal such fold.

```
>>> S.sum $ yield 1 >> yield 2
3 :> ()

>>> S.length $ yield 1 >> yield 2
2 :> ()

>>>  S.minimum $ yield 1 >> yield 2
Just 1 :> ()

>>> S.toList $ yield 1 >> yield 2
[1,2] :> ()
```

The underscored variants of standard folds like `S.sum_`, `S.product_` and so on drop the final return value. In simple cases like those above, we could as well have used them:

```
>>> S.sum_ $ yield 1 >> yield 2
3

>>> S.toList_ $ yield 1 >> yield 2
[1,2]
```

because above we were exiting streaming for good, and the streams we were folding just returned `()` anyway. We will quickly see why the 'official' versions of these simple folds preserve the final return value of the folded stream.

The glue that holds together a typical Haskell pair, or our left-strict variant, is the glue that holds together the *successive phases* of a `Stream (Of a) m r`, linking each yielded element with the rest of the stream. That is, when we inspect a stream of the type

```
Stream (Of a) m r
```

we enter the monad of effects, e.g. `IO`, and there we either immediately hit upon the return or exit value - something of type `r`. In which case we are done streaming. But we may also hit upon something of the type `Of a (Stream (Of a) m r)`. In that case we are "still streaming." The pattern for this case, if we are dealing with it directly, looks like so

```
a :> rest
```

The first member of the pair - an individual `a` - is the present stream element, and it is linked by the pairing constructor to *the rest of the stream.*

The function `next`

```
next :: Monad m => Stream (Of a) m r -> m (Either r (a, Stream (Of a) m r))
```

expresses these possibilities in terms of the standard base types `(,)` and `Either`. So to elimate all of the elements of a stream in order to get the return value, we might write:

```
forget :: Monad m => Stream (Of a) m r -> m r
forget str = do
   e <- next str
   case e of
     Left r -> return r
     Right (a, rest) -> forget rest
```

(This is `Streaming.Prelude.effects`.) I will return to `next` in a moment. We are at present only interested in the construction of a `Stream (Of a) ...`

Note that `Stream` has a show instance which will work when we specialize `m` to `Identity`:

```
>>> each [1..4] :: Stream (Of Int) Identity ()
Step (1 :> Step (2 :> Step (3 :> Step (4 :> Return ()))))
```

This reveals the constructors which are hidden in the `Internal` module (here `Step` and `Return`) but we can observe the similarity to the construction of a Haskell list. The principal operational difference (apart from the fact that it has a final "return" value - here the vestigial `()`) is that a `Stream (Of Int) Identity ()` is strict in its leaves. We can change that by moving from `Of Int` to `(,) Int`:

```
>>> :t lazily
lazily :: Of a r -> (a, r)
>>> maps lazily (each [1..4]) :: Stream ((,) Int) Identity ()
Step (1,Step (2,Step (3,Step (4,Return ()))))
```