Monads – from StackOverflow

https://stackoverflow.com/questions/44965/what-is-a-monad?rq=1

First: The term **monad** is a bit vacuous if you are not a mathematician. An alternative term is **computation builder** which is a bit more descriptive of what they are actually useful for.
You ask for practical examples:

Example 1: List comprehension:

```
[x*2 | x<-[1..10], odd x]
```

This expression returns the doubles of all odd numbers in the range from 1 to 10. Very useful!

It turns out this is really just syntactic sugar for some operations within the List monad. The same list comprehension can be written as:

```
do
  x <- [1..10]
  if odd x
     then [x * 2]
     else []
```

Or even:

```
[1..10] >>= (\x -> if odd x then [x*2] else [])
```

Example 2: Input/Output:

```
do
  putStrLn "What is your name?"
  name <- getLine
  putStrLn ("Welcome, " ++ name ++ "!")
```

Both examples use monads, AKA computation builders. The common theme is that the monad *chains operations* in some specific, useful way. In the list comprehension, the operations are chained such that if an operation returns a list, then the following operations are performed on *every item* in the list. The IO monad on the other hand performs the operations sequentially, but passes a "hidden variable" along, which represents "the state of the world", which allows us to write I/O code in a pure functional manner.
It turns out the pattern of *chaining operations* is quite useful and is used for lots of different things in Haskell.
Another example is exceptions: Using the `Error` monad, operations are chained such that they are performed sequentially, except if an error is thrown, in which case the rest of the chain is abandoned.
Both the list-comprehension syntax and the do-notation are syntactic sugar for chaining operations using the `>>=` operator. A monad is basically just a type that supports the `>>=` operator.

Example 3: A parser

This is a very simple parser which parses either a quoted string or a number:

```
parseExpr = parseString <|> parseNumber

parseString = do
      char '"'
      x <- many (noneOf "\"")
      char '"'
      return (StringValue x)

parseNumber = do
   num <- many1 digit
   return (NumberValue (read num))
```

The operations `char`, `digit`, etc. are pretty simple. They either match or don't match. The magic is the monad which manages the control flow: The operations are performed sequentially until a match fails, in which case the monad backtracks to the latest `<|>` and tries the next option. Again, a way of chaining operations with some additional, useful semantics.

Example 4: Asynchronous programming

The above examples are in Haskell, but it turns out F# also supports monads. This example is stolen from Don Syme:

```
let AsyncHttp(url:string) =
   async { let req = WebRequest.Create(url)
           let! rsp = req.GetResponseAsync()
           use stream = rsp.GetResponseStream()
           use reader = new System.IO.StreamReader(stream)
           return reader.ReadToEnd() }
```

This method fetches a web page. The punch line is the use of `GetResponseAsync` - it actually waits for the response on a separate thread, while the main thread returns from the function. The last three lines are executed on the spawned thread when the response have been received.

In most other languages you would have to explicitly create a separate function for the lines that handle the response. The `async` monad is able to "split" the block on its own and postpone the execution of the latter half. (The `async {}` syntax indicates that the control flow in the block is defined by the `async` monad.)

How they work

So how can a monad do all these fancy control-flow thing? What actually happens in a do-block (or a *computation expression* as they are called in F#), is that every operation (basically every line) is wrapped in a separate anonymous function. These functions are then combined using the `bind`operator (spelled `>>=` in Haskell). Since the `bind` operation combines functions, it can execute them as it sees fit: sequentially, multiple times, in reverse, discard some, execute some on a separate thread when it feels like it and so on.

As an example, this is the expanded version of the IO-code from example 2:

```
putStrLn "What is your name?"
>>= (\_ -> getLine)
>>= (\name -> putStrLn ("Welcome, " ++ name ++ "!"))
```

This is uglier, but it's also more obvious what is actually going on. The `>>=` operator is the magic ingredient: It takes a value (on the left side) and combines it with a function (on the right side), to produce a new value. This new value is then taken by the next `>>=` operator and again combined with a function to produce a new value. `>>=` can be viewed as a mini-evaluator.

Note that `>>=` is overloaded for different types, so every monad has its own implementation of `>>=`. (All the operations in the chain have to be of the type of the same monad though, otherwise the `>>=`operator won't work.)

The simplest possible implementation of `>>=` just takes the value on the left and applies it to the function on the right and returns the result, but as said before, what makes the whole pattern useful is when there is something extra going on in the monad's implementation of `>>=`.

There is some additional cleverness in how the values are passed from one operation to the next, but this requires a deeper explanation of the Haskell type system.

Summing up

In Haskell-terms a monad is a parameterized type which is an instance of the Monad type class, which defines `>>=` along with a few other operators. In layman's terms, a monad is just a type for which the `>>=` operation is defined.

In itself `>>=` is just a cumbersome way of chaining functions, but with the presence of the do-notation which hides the "plumbing", the monadic operations turns out to be a very nice and

useful abstraction, useful many places in the language, and useful for creating your own mini-languages in the language.

Why are monads hard?

For many Haskell-learners, monads are an obstacle they hit like a brick wall. It's not that monads themselves are complex, but that the implementation relies on many other advanced Haskell features like parameterized types, type classes, and so on. The problem is that Haskell I/O is based on monads, and I/O is probably one of the first things you want to understand when learning a new language - after all, it's not much fun to create programs which don't produce any output. I have no immediate solution for this chicken-and-egg problem, except treating I/O like "magic happens here" until you have enough experience with other parts of language. Sorry.

Excellent blog on monads: http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html