ELIXIR MODULES

Aamer F Rakla

67-495 – Advanced Topics in Information Systems/Innovation in Information Systems

WHAT IS A MODULE?

- Simply, it is a group of functions; It is similar to a class in an OOP Language
- Modules are used without even realizing it, such as the String module
- The `defmodule` macro is used to created modules
- Elixir also allows for nested modules

```
iex> defmodule Math do
...>   def sum(a, b) do
...>     a + b
...>   end
...> end

iex> Math.sum(1, 2)
3
```

```
defmodule Example do
  def greeting(name) do
    "Hello #{name}."
  end
end

iex> Example.greeting "Sean"
"Hello Sean."
```

```
defmodule Example.Greetings do
  def morning(name) do
    "Good morning #{name}."
  end

  def evening(name) do
    "Good night #{name}."
  end
end

iex> Example.Greetings.morning "Sean"
"Good morning Sean."
```

COMPILATION

- Modules are normally written into files
- This way they can be compiled and reused
- It can be compiled using `elixirc math.ex` resulting in a BEAM file in bytecode for Eralang VM
- Running `iex` in the same directory as the BEAM file will make it available

```
defmodule Math do
  def sum(a, b) do
    a + b
  end
end
```

```
iex> Math.sum(1, 2)
3
```

COMPILATION

- Elixir projects are usually organized into three directories:
  - ebin - contains the compiled bytecode (usually .beam files)
  - lib - contains elixir code (usually .ex files)
  - test - contains tests (usually .exs files)
- For actual projects, a build tool called `mix` handles compiling

SCRIPTED MODE

- Scripted mode is used primarily for learning purposes as well as testing
- Elixir scripts are files ending in `.exs`
- Elixir treats `.ex` and `.exs` exactly the same, except `.exs` isn't compiled

```
defmodule Math do
  def sum(a, b) do
    a + b
  end
end

IO.puts Math.sum(1, 2)
```

NAMED FUNCTIONS

- Functions can be defined inside of a module using `def` or `defp` for private
- Public functions can be called from other modules and private ones only from within the module

```
defmodule Math do
  def sum(a, b) do
    do_sum(a, b)
  end

  defp do_sum(a, b) do
    a + b
  end
end
```

```
IO.puts Math.sum(1, 2)    #=> 3
IO.puts Math.do_sum(1, 2) #=> ** (UndefinedFunctionError)
```

NAMED FUNCTIONS

- Like in Java and other languages, Elixir supports function overloading
  - In other words, having functions with the same name but different arities
- Elixir also supports guards for functions

```
defmodule Math do
  def zero?(0) do
    true
  end

  def zero?(x) when is_integer(x) do
    false
  end
end
```

```
IO.puts Math.zero?(0)          #=> true
IO.puts Math.zero?(1)          #=> false
IO.puts Math.zero?([1, 2, 3]) #=> ** (FunctionClauseError)
IO.puts Math.zero?(0.0)        #=> ** (FunctionClauseError)
```

DEFAULT ARGUMENTS

- Functions support default arguments
- Expressions can be used as default arguments
  - They are only evaluated when called, not during definition

```
defmodule DefaultTest do
  def dowork(x \\ IO.puts "hello") do
    x
  end
end
```

```
iex> DefaultTest.dowork
hello
:ok
iex> DefaultTest.dowork 123
123
iex> DefaultTest.dowork
hello
:ok
```

DEFAULT ARGUMENTS

- When overloading a function (when it has multiple clauses), a function without a body (a function head), is needed to define default values

```elixir
defmodule Concat2 do
  def join(a, b \\ nil, sep \\ " ")

  def join(a, b, _sep) when is_nil(b) do
    a
  end

  def join(a, b, sep) do
    a <> sep <> b
  end
end
```

```elixir
IO.puts Concat2.join("Hello", "world")      #=> Hello world
IO.puts Concat2.join("Hello", "world", "_") #=> Hello_world
IO.puts Concat2.join("Hello")               #=> Hello
```

DEFAULT ARGUMENTS

- When overloading a function (when it has multiple clauses), the order they are listed makes a difference
- Elixir/Erlang will use the first matching function
- The compiler will give a warning when this happens

MODULE ATTRIBUTES

- Module attributes originate from Erlang
- Module attributes in Elixir serve three purposes:
  - They serve to annotate the module, often with information to be used by the user or the VM.
  - They work as constants.
  - They work as a temporary module storage to be used during compilation.

ANNOTATIONS

- Annotations are used for many purposes, such as tracking the version of a module
- Erlang VM uses annotations to track the version of a module
  - If there is no version, the MD5 checksum of the module is used
- Elixir has a number of reserved attributes, inlcuding:
  - `@moduledoc` - provides documentation for the current module.
  - `@doc` - provides documentation for the function or macro that follows the attribute.
  - `@behaviour` - (notice the British spelling) used for specifying an OTP or user-defined behaviour.
  - `@before_compile` - provides a hook that will be invoked before the module is compiled. This makes it possible to inject functions inside the module exactly before compilation.

ANNOTATIONS

- `@moduledoc` and `@doc` are the most used attributes and Elixir expects everyone to use them
- Elixir treats documentation as first-class and provides many functions to access it
- Elixir promotes useing markdown with heredocs to write readable documentation
- Heredocs are multiline strings, starting and ending with “”””

```
defmodule Math do
  @moduledoc """
  Provides math-related functions.

  ## Examples

      iex> Math.sum(1, 2)
      3

  """

  @doc """
  Calculates the sum of two numbers.
  """
  def sum(a, b), do
```

ANNOTATIONS

- There is a tool called ExDoc for generating HTML pages from the documentation
- Attributes can also be used by developers or extended by libraries to support custom behavior

SOURCES

- These slides were made using information and examples from the official Elixir Language website and Elixir School
  - http://elixir-lang.org/getting-started/modules.html
  - http://elixir-lang.org/getting-started/module-attributes.html
  - https://elixirschool.com/lessons/basics/modules/