DATA STRUCTRES IN ELIXIR

DIFFERENT DATA STRUCTURES IN THESE SLIDES WILL BE

- MAPS
- TUPLES
- LISTS
- KEYWORD LISTS
- STRUCTS

MAPS

- MAPS IN ELIXIR ARE A 'KEY-VALUE' DATA STRUCTURE WHICH ARE CREATED WITH `%{}` SYNTAX

```
iex(1)> map = %{"a" => 2, "b" => 5, :n => :seven}
%{:n => :seven, "a" => 2, "b" => 5}
iex(2)> map["a"]
2
iex(3)> map[:n]
:seven
```

TUPLES

- TUPLES ARE A DATA STRUCTURE WHICH STORE ELEMENTS IN ONE CONTIGUOUS BLOCK OF MEMORY. ACCESSING TUPLES IS VERY EFFICENT BECAUSE YOU CAN ACSESS THEM THROUGH INDEX WHERE THE INDEX STARTS FROM 0

```
iex(1)> tuple = {:ok, :hello, :error, 13, "alright"}
{:ok, :hello, :error, 13, "alright"}
iex(2)> elem(tuple, 4)
"alright"
iex(3)> elem(tuple, 0)
:ok
```

LISTS

- LISTS IN ELIXIR ARE SETUP AS A LINKED DATA STRUCTURE WHICH MEANS TO ACCESS A RANDOM ELEMENT YOU MUST ITERATE THROUGH EVERY ELEMENT BEFORE IT WHICH MAKES LISTS EASY TO PERFORM RECURSION ON
- LISTS ARE SEPEARTED INTO TWO PARTS THE HEAD AND TAIL WHERE THE HEAD IS THE FIRST ELEMENT OF THE LIST AND TAIL IS THE REST OF THE LIST

```
iex(5)> list = [1, 2, 3]
[1, 2, 3]
iex(6)> hd(list)
1
iex(7)> tl(list)
[2, 3]
```

- THIS SEPERATION ALLOWS US TO PERFORM SOMETHING CALLED TAIL RECURSION LETS CREATE AN EXAMPLE USING THIS.
  - WE WILL CREATE A FUNCTION WHICH WILL SUM UP THE ELEMEMNTS OF NUMBERED LIST
    SO FIRST CREATE A FILE NAMED `lists.exs` THIS IS A SCRIPT FILE WHICH WON'T BE COMPILED BUT SIMPLY RAN

- START BY CREATING A MODULE CALLED LISTS WITH A FUNCTION SUM(LIST) WHICH WILL CALL A PRIVATE SUM FUNCTION WITH THE LIST AND A STARTING SUM OF 0

```
defmodule Lists do

    def sum(list) do
        do_sum(list, 0)
    end

end
```

- NEXT WE WILL CREATE 2 PRIVATE FUNCTIONS CALLED DO_SUM WITH 2 ARGUMENTS THE HEAD AND TAIL OF A LIST AND SUM WHICH WILL CALL ITSELF WITH THE TAIL OF THE LIST WHILE ADDING THE HEAD TO THE SUM.
- AND ANOTHER PRIVATE FUNCTION THAT TAKES THE EMPTY LIST

```elixir
defp do_sum([head | tail], sum) do
    do_sum(tail, head+sum)
end


defp do_sum([], sum) do
    sum
end
```

```elixir
defmodule Lists do

  def sum(list) do
    do_sum(list, 0)
  end

  defp do_sum([head | tail], sum) do
    IO.puts(sum)
    do_sum(tail, sum + head)
  end

  defp do_sum([], sum) do
    sum
  end
end

list = [1, 2, 3, 4, 5, 6, 7]

output = list |> Lists.sum()

IO.puts output
```

NOW SAVE YOUR FILE AND IN YOUR TERMINAL TYPE `elixir lists.exs` AND IT SHOULD OUTPUT

```
@user% elixir lists.exs
0
1
3
6
10
15
21
28
```

EXPANDING FROM THAT SAME CONCEPT WE COULD DO THE SAME WITH A LIST OF WORDS!

```
def concat(list) do
    list = Enum.reverse(list)
    do_concat(list, "")
  end

  def do_concat([head | tail], word) do
    do_concat(tail, head<>" "<>word)
  end

  def do_concat([], word) do
    word
  end
```

DUE TO RECURSION WE MUST USE ENUM.REVERSE WHICH SWAPS THE ORDER OF THE ELEMENTS IN THE LIST TO PRINT OUT WHAT WE WANT PROPERLY.

NOW WE CAN TEST OUT OUR NEW FUNCTION IN IEX

```
iex(1)> list = ["I", "love", "learning", "elixir!"
...(1)> ]
["I", "love", "learning", "elixir!"]
iex(2)> Lists.concat(list)
"I love learning elixir! "
```

KEYWORD LISTS

- KEYWORD LISTS ARE A SPECIAL TYPE OF LIST WHERE EACH ELEMENT IS A TWO ELEMENT TUPLE WITH THE FIRST ELEMENT OF THE TUPLE BEING AN ATOM

```
iex> list = [{:elixir, 1}, {:phoenix, 2}]
[elixir: 1, phoenix: 2]
list == [elixir: 1, phoenix: 2]
true
```

STRUCTS

- STRUCTS ARE EXTENSIONS OF MAPS, STRUCTS TAKE THE NAMES OF THEIR MODULE
  THE SYNTAX OF ACCESSING A STRUCT IS `%MODULE_NAME{}`
- WE COULD DEFINE BY CALLING DEFSTRUCT THEN ADDING ELEMENTS

```
defmodule Test do
    defstruct language: "elixir", passion: "programming"
end

iex> temp = %Test{}
iex> temp.language
"elixir
```