```
    # < > <= >= are also comparison operators
```

Control structures

```
# ---------- CONTROL STRUCTURE ----------

# CONDITIONALS

# IF ELSE
    # elixir has no built-in implementation of elseif statements
    # do end => used to mark the start and end of the if else block

if false do
    "watermelon"
else
    "shit ass"
end # this evaluates to "shit ass"

# UNLESS ELSE
    # unless => provides for the negation of a specified condition
    # do end => used to mark the start and end of the unless else block

unless true do
  "this will never be seen"
else
  "this will shit"
end # this evaluates to "this will shit"

# PATTERN MATCHING
    # elixir's powerful pattern-matching construct rivals Rust in its conciseness and completeness
    # case => declares and creates a case statement, similar to switch case statements in other languages
    # do end => used to mark the start and end of a given case statement's cases, within which -> speciifes the relationship between a given case condition and the internal logic to run if said condition is satisfied
    # _ => catch-all operator used as the equivalent of a default statement in other languages

case {:one, :two} do
    {:four, :five} ->
        "this won't match"
    {:one, x} ->
        "this will match and bind `x` to `:two` in this clause"
    _ ->
        "this will match any value"
end # notice that pattern-matching can occur for tuples and other data structures

[head | _] = [1,2,3] # note the catch-all operator _ can be used to throw away any unwanted value, as seen here where only the head value is matched and assigned and the tail is thrown away
head # this evaluates to the integer value 1

# COND
    # cond => declares and creates a cond block, which runs mutliple conditional checks at the same time, equivalent to switch case statements in other languages and often used within elixir as a concise alternative to nesting mutliple if statements, with -> specifying the relationship between a condition and the internal logic to run if a given condition evaluates to true
    # do end => used to mark the start and end of the cond block
    # true => it is convention to set the last condition as true to act as a default statement within a cond block

cond do
  1 + 1 == 3 ->
    "I will never be seen"
  2 * 5 == 12 ->
    "me neither"
  1 + 2 == 3 ->
    "but I will"
end # this evaluates to "but I will"

cond do
  1 + 1 == 3 ->
    "I will never be seen"
  2 * 5 == 12 ->
    "me neither"
  true ->
    "but I will"
end # this evaluates to "but I will" as well due to the presence of the true condition which acts as a default statement

# TRY CATCH AFTER
    # try catch => declares a try catch block, similar to try except in other languages
    # after => specifies code that should execute regardless of whether a value is caught by the try catch block
    # rescue => used to handle specified errors
    # do end => used to mark the start and end of a try catch after block

try do
  throw(:hello)
catch
  message -> "got #{message}."
after
  IO.puts("I'm the after clause.")
end # this prints "I'm the after clause" to the stdout

# RANGES
    # .. => creates an inclusive range on both ends

1..10 # this evaluates to a range that stores integers from 1 to 10

# LOOPS
    # as a functional language, elixir does not have conventional imperative loops implemented, but offers higher-order functions, recursion and list comprehension that allow for the same effect in a concise manner
    # Enum module => provides Enum.each, Enum.map, Enum.reduce and other higher-order functions

# LIST COMPREHENSION
    # uses the syntax => for {PATTERN} <- {ITERABLE STRUCTURE}, {FILTER CONDITIONS}, do: {EXPRESSION}
        # pattern => applies a specified pattern-matching construct against elements from the iterable structure
        # iterable structure => data structure like a range, list etc
        # filter conditions => optional conditions to further filter elements
        # expression => does something to the given element and includes it in the resulting new list

numbers = [1, 2, 3, 4, 5]
doubled_numbers = for n <- numbers, do: n * 2 # this iterates over the list numbers, taking each value and multiplying it by 2, then reassigning it to a new list doubled_numbers
doubled_numbers # this evaluates to the integer list of value [2, 4, 6, 8, 10]
```