Course Content

udemy.com/elixir-scalable-and-efficient-application-development/learn/v4/t/lecture/6958186

Elixir processes are self-contained abstractions. The context of each process is isolated from the contexts of other processes. Messages are required to share information between the processes. This is considered the actor-model. Each process is an actor, capable of sending and receiving messages from other actors. Based on the contents of a message, an actor may perform certain actions. This is the foundation of Elixir processes: self-contained actors operate on information sent to them, and the result is often sent back to the calling process.

Of course, if that's all there was to it, we would be essentially done. But concurrent programming is never that simple.

There are several functions automatically available to most Elixir modules and inside an `iex` session, and these will be the majority of the discussion of this chapter.

Self

We have seen process identifiers before, but now it's time to explain the numbers involved with the numbers shown.

If we start an interactive session and type `self`, we will get some response that may look similar to the following:

```
1. iex(1)> self
2. #PID<0.60.0>
```

The response to the `self/0` function is a process identifier. This is the identity of the current process (the REPL, in this case). The numbers in the identifier are the process address. The first number, `0` in this case, tells us the Erlang node the process exists on. This will always be `0`, the special address for the local node. The second number, `60`, in this case, is the first 15 bits of the process number, part of the process index. Finally, the last number, `0`, in this case, is the rest of the process index address, typically 16-18 bits. These three numbers give us the full address to any process. We can use this reference identifier as an address to send messages.

Sending messages

Intuitively, we can use the `send/2` message to send messages between the processes. For now, we can send messages to ourselves to be later received:

```
1. iex(1)> me = self
2. #PID<0.60.0>
3. iex(2)> send me, :ping
4. :ping
```

We created a reference to ourselves, called it `me`, and then we sent a `:ping` atom to ourselves using `send/2`. The `:ping` atom that is returned is simply the message of `send/2` being evaluated locally. We are not sending computation, we are sending the result of some computation.

Sending messages turns out to be very easy. Receiving messages is generally easy, but there are cases to look out for.

Receiving messages

To receive the message we sent ourselves, we need to use the `receive` block. Perhaps unsurprisingly, the `receive` block may feel like a special form of case, where the variable being matched against is sent from another process (or ourselves!). Let's receive our ping:

```
1. iex(3)> receive do
2. ...(3)> x -> IO.puts("#{inspect x}")
3. ...(3)> end
4. :ping
5. :ok
```

We receive the atom, `:ping`, and the expression of printing the atom returns `:ok`. This is pattern matching showing its beautiful self again. However, in this case, if any other process sent another message **before** we sent `:ping`, we would receive that message first. The message is simply queued.

Sending a message to a process doesn't interfere with the process it is sent to, it is placed in that process' message queue. There, the message will sit until the process decides to check its queue. If the receiving process never checks the queue and many messages are sent, the process can and will crash because messages take space, the space that is never reclaimed during the process' lifetime. The size of the message queue is fairly large and the number of messages allowed to be queued does depend on the size of each message as well.

Since we can queue a number of messages before receiving them, we can create a process that talks to itself:

```
1. iex(4)> 1..5 |> Enum.map(&(send(me, &1 * &1)))
2. [1, 4, 9, 16, 25]
3. iex(5)> receive do
4. ...(5)> x -> x
5. ...(5)> end
6. 1
```

Here, we sent the first five squares to ourselves and we received the first one. We can keep going and retrieve the next four squares; however, this gets incredibly tedious in its current form. Thus, the `flush/0` helper function may be very useful when testing message passing in the interactive session. We can dump the rest of the messages in the current process mailbox:

```
iex(6)> flush()
4
9
16
25
:ok
```

Plus, a process that is able to talk to itself via `send/2` and `receive` isn't terribly exciting. Let's look at creating our own processes!

Spawn

The Elixir `Kernel` module provides us with the `spawn/1` and `spawn/3` functions for creating processes. These can be used to create separate processes that compute some result and send back the result, or really, can perform any sort of work we can imagine.

The `spawn/1` function takes a zero argument function and executes the function inside a new process:

```
iex(7)> spawn fn -> 6 * 7 end
#PID<0.95.0>
```

Notice that the process identifier of the new process is returned, not the result of the computation. Furthermore, using the `Process.alive?/1` function, we can see that the process is actually dead:

```
iex(8)> pid = spawn fn -> 6 * 7 end
#PID<0.97.0>
iex(9)> Process.alive?(pid)
false
```

The process exits as soon as the function returns. The spawned process has nothing left to do; the result is dropped and the process and its context is marked for cleanup and is discarded. If a process exits for a non-normal reason, an error is raised, for example, it does not affect or notify the parent process:

```
iex(10)> spawn fn -> raise :oops end
23:45:55.643 [error] Process #PID<0.102.0> raised an exception
** (RuntimeError) oops
    :erlang.apply/2
```

The message is logged and life goes on.

If we want the result of a spawned process, we will need to tell the process to send it back when it's done computing it:

```
iex(1)> parent = self()
#PID<0.60.0>
iex(2)> spawn fn -> send(parent, 6 * 7) end
#PID<0.63.0>
iex(3)> receive do
...(3)> x -> IO.puts("#{inspect x}")
...(3)> end
42
:ok
```

We create a reference to the parent (current) process, and we spawn a new process that sends the result of `6 * 7` to the parent process. Notice that we can't use `self/0` in-line here because it would be evaluated to a different process ID if we did. Finally, from the parent, we receive the result and print it to standard out.

Again, these sort of manual steps of sending messages between the processes is tedious and unwieldy though. Thus, we use the `spawn/3` function.

Both `spawn/1` and `spawn/3` are in-lined by the compiler. Furthermore, Erlang only has a `spawn/3` function, and the parameters to Elixir's `spawn/3` are the same for Erlang; and it was done this way for consistency between the languages.

The `spawn/3` function requires us to specify the function in the older Erlang syntax, but allows us to better create the processes that do more useful computations.

The older Erlang syntax is to specify the module, function, and arguments, even if none, in the form of a triple. That is, if we wanted to launch the `do_work/0` function of the `Worker` module, we would pass the following to `spawn/3`:

```
1. spawn(Worker, :do_work, [])
```

Fortunately, this is consistent throughout the spawning functions.

Let's actually create the `Worker` module and have it compute our squares for us.

In a file, `worker.exs`, we can define the following module and single function:

```
1. defmodule Worker do
2.   def do_work() do
3.     receive do
4.       {:compute, x, pid} ->
5.         send pid, {:result, x * x}
6.     end
7.     do_work()
8.   end
9. end
```

Our function is relatively uninteresting, but we are more interested in the basic structure for the moment.

Once started, our function waits for a message in the form of a triple, `{:compute, x, pid}`, where `:compute` tags the message, `x` is the number we wish to square, and `pid` is the sending process ID to send the return result. Once this process receives a triple, it sends the result to the calling process and launches into an infinite loop.

Tagging messages is common practice so that we can easily distinguish the message, how to parse, what action to perform, and others.

Infinitely looping like this is actually more common than you might think. Many languages and frameworks behave this way. Elixir, however, makes this explicit to the programmer.

In an interactive session, we can import the module and spawn it in another thread:

```
1. iex(1)> import_file "worker.exs"
2. {:module, Worker,
3.  <<70, 79, 82, 49, 0, 0, 5, 52, 66, 69, 65, 77, 69, 120, 68, 99, 0, 0, 0, 96,
   131, 104, 2, 100, 0, 14, 101, 108, 105, 120, 105, 114, 95, 100, 111, 99, 115,
   95, 118, 49, 108, 0, 0, 0, 2, 104, 2, ...>>,
4.  {:do_work, 0}}
5. iex(2)> pid = spawn(Worker, :do_work, [])
6. #PID<0.67.0>
7. iex(3)> Process.alive?(pid)
8. true
```

Our worker process is waiting for another process to provide it work; let's give it something to do:

```
1. iex(4)> send pid, {:compute, 4, self()}
2. {:compute, 4, #PID<0.60.0>}
```

We should now have the result in our inbox:

```
1. iex(5)> flush()
2. {:result, 16}
3. :ok
```

Furthermore, checking to see if our worker process is still alive should yield that it is in fact still running and should still be accepting more work:

```
1. iex(6)> Process.alive?(pid)
2. true
3. iex(7)> send(pid, {:compute, 16, self()})
4. {:compute, 16, #PID<0.67.0>}
5. iex(8)> flush()
6. {:result, 256}
7. :ok
```

The `Worker` module isn't very resilient, though. It's trivial to think of an example that will bring it down:

```
1. iex(9)> send(pid, {:compute, "square this", self()})
2. {:compute, "square this", self(
3. 17:42:10.396 [error] Process #PID<0.67.0> raised an exception
4. ** (ArithmeticError) bad argument in arithmetic expression
5.     iex:6: Worker.do_work/0
6. iex(10)> Process.alive?(pid)
7. false
```

In this case, we might actually want to be notified that the process exited so that we know not to attempt to send messages to it or to resurrect it before sending more messages. We could spend some time thinking of a way to do this with just `spawn/3` or we could look at `spawn_link/3`, which solves this problem for us.

Process links

Process links create relationships between the processes and enable another channel of communication to be used between the two. The channel allows processes to be signaled when another process dies. It might not be obvious that cascading process death would be

useful, but it turns out to be very useful, especially with respect to the fail-fast philosophy.

For complicated systems, there will be many processes working together to model and compose the system. Many of these processes will be interrelated and possibly codependent on each other for results. When a process dies, because there is no longer a question of if, the programmer will have to make a decision: should the programmer attempt to conceive of all the possibilities and state when a process dies, or kill all the dependent processes and restart the failed processes in a clean state?

The former strategy is an alluring trap and often chosen for naive reasons such as the number of failure cases is thought to be small or that certain things, for, example, hardware failure or some other seemingly uncommon cause, will never happen. However, many of these reasons for choosing this route are based on assumptions that are, at best, misguided.

The overall design of the system becomes easier when the assumptions are limited, and the steps to take on failure are clear. That is, the system becomes easier to reason about, and when the assumptions are limited and clear, the failures explicit.

Process links serve the purpose of enabling the cascading failure of dependent or downstream processes when an upstream process fails.

Spawning with links

Instead of spawning a child process and then adding a link between the parent process and the child process, we could spawn the child process with the link already established. This is accomplished with `spawn_link/1` and `spawn_link/3`.

Using `spawn_link/1` in place of `spawn/1` works the same as you expect; in fact, they are no different if we use the same example as well:

```
1. iex(1)> spawn_link fn -> 6 * 7 end
2. #PID<0.62.0>
```

However, the difference of creating a link right away is that the parent process is notified if the child process fails during its startup steps. For example, if we spawn a child process that immediately fails, the error is propagated correctly to the parent process:

```
 1. iex(2)> spawn_link fn -> raise "failing" end
 2. #PID<0.62.0>
 3. iex(2)> spawn_link fn -> raise "failing" end
 4. ** (EXIT from #PID<0.60.0>) an exception was raised:
 5.     ** (RuntimeError) failing
 6.         :erlang.apply/2
 7. Interactive Elixir (1.0.5) - press Ctrl+C to exit (type h() ENTER for help)
 8. 23:34:05.758 [error] Process #PID<0.64.0> raised an exception
 9. ** (RuntimeError) failing
10.     :erlang.apply/2
11. iex(1)>
```

Note that your output order may differ.

A linked process doesn't always kill the parent, however. The propagation of the errors is related to the **reason** the child processing is terminating.

For example, we can create a more robust version of our worker process from the last section to also receive the exit signals. The new version might look similar to the following code:

```
1.  defmodule Worker do
2.    def do_work() do
3.      receive do
4.        {:compute, x, pid} ->
5.          send pid, {:result, x * x}
6.        {:exit, reason} ->
7.          exit(reason)
8.      end
9.      do_work()
10.   end
11. end
```

Save the preceding code in either a new version of `worker.exs` or the same version as earlier. Then, we could import and spawn it in an interactive session:

```
1. iex(1)> import_file "worker.exs"
2. {:module, Worker,
3.  <<70, 79, 82, 49, 0, 0, 5, 160, 66, 69, 65, 77, 69, 120, 68, 99, 0, 0, 0, 96,
   131, 104, 2, 100, 0, 14, 101, 108, 105, 120, 105, 114, 95, 100, 111, 99, 115,
   95, 118, 49, 108, 0, 0, 0, 2, 104, 2, ...>>,
4.  {:do_work, 0}}
```

We will be using the `spawn_link/3` function instead of `spawn_link/1`. The former requires the same syntax as `spawn/3`, so there is nothing surprising about the next line of code:

```
1. iex(2)> pid = spawn_link Worker, :do_work, []
2. #PID<0.67.0>
```

Of course, we can still send work to the process using the usual message:

```
1. iex(3)> send pid, {:compute, 4, self}
2. {:compute, 4, #PID<0.60.0>}
3. iex(4)> flush
4. {:result, 16}
5. :ok
```

But we can also kill the process without causing a failure in the parent process:

```
1. iex(5)> Process.alive?(pid)
2. true
3. iex(6)> send pid, {:exit, :normal}
4. {:exit, :normal}
5. iex(7)> Process.alive?(pid)
6. false
```

The child process was running and then we sent the `exit` signal. After that, the child process was no longer running and the parent process was left unaffected. However, we could have sent a strong `exit` signal since we are requesting the reason for exiting, and therefore, such a reason can bubble up to the parent process. There are two standard exit reasons in Elixir, `:normal` and `:kill`. Any other atom can also be used, and there are few processing rules for how they are handled. For more information on this, see the help of `Process.exit/2`.

Process monitor

Process monitoring is slightly different from process links, but are fairly similar. Monitors are special, stackable, and unidirectional links.

Links were bidirectional and facilitated the cascading death of many processes. Two processes where the failure of one makes the other useless is a great example of where using a process link is ideal. However, maybe a process simply needs to know the state of another process and not necessarily the other way around. Furthermore, since monitors are stackable, removing a monitor doesn't remove all of the other monitors. Unlinking two or more processes will do exactly this; it will remove every link, in a cascade, ruining the assumptions the links provide.

This is exactly where process monitors stand apart from process links. They enable the monitoring process to receive messages about the state of the monitored process. To accomplish this, we use `spawn_monitor/1` and `spawn_monitor/3`. These work exactly as you might expect, same as `spawn` and `spawn_link`. However, instead of simply returning the PID of the child process, they return an atom of the child PID and the reference to the monitor. The monitor reference can be used for removing the monitor, if need be:

```
1. iex(1)> {pid, _} = spawn_monitor(fn -> :timer.sleep(500) end)
2. {#PID<0.64.0>, #Reference<0.0.3.82>}
3. iex(2)> Process.alive?(pid)
4. false
5. iex(3)> flush
6. {:DOWN, #Reference<0.0.3.82>, :process, #PID<0.64.0>, :normal}
7. :ok
```

Using a process monitor is very similar to using a link, however. Instead of a process failure or normal termination resulting in a cascading termination of processes, the monitoring process receives a regular message in its inbox about the death of the monitored process. The monitoring process can receive this message and is given the opportunity to perform some action, based on the receipt of these messages.

Storing state inside processes

Elixir processes are great. Well, they are great, so far, as long as you don't need them to remember anything. This quickly becomes a problem when we attempt to extend the uses of the Elixir processes and we need a way for a process to remember or store data.

It turns out, though, this is relatively easy to solve with `spawn/3` and friends. The third argument of the three arity versions could be considered the initial state of the process. The process, then, could begin running using its initial state. Then the question becomes, how

does it update its state? Elixir data is immutable, so how does the process modify this data? The answer is in the tail-recursive infinite loop. In each invocation of the loop, the state of the process is passed, with changes and all, to itself.

Let's create a simple key-value storage process to demonstrate this. The overall design will be a module that accepts the `:put` and `:get` messages:

```
1.  defmodule KV do
2.    def start_link do
3.      spawn_link(fn -> loop(%{}) end)
4.    end
5.    defp loop(map) do
6.      receive do
7.        {:put, key, value, sender} ->
8.          new_map = Map.put(map, key, value)
9.          send sender, :ok
10.         loop(new_map)
11.       {:get, key, sender} ->
12.         send sender, Map.get(map, key)
13.         loop map
14.     end
15.   end
16. end
```

The `start_link` function is a convenience function for spawning our `KV` process; it simply starts the looping portion with an initially empty map. The preceding `receive do` loop is fairly straightforward. We match against two different kinds of messages, `:put` and `:get`. If we are given a `:put` message, we update the map (our internal state) with the new key using the `Map.put/3` function and then we send the calling process, the `:ok` message. Finally, we recurse using the updated map. If we are given a `:get` message, we send the value, or more precisely, the result of `Map.get/2` to the calling process and loop with the existing map.

Save the `module` definition to a file and let's try it out in an interactive session:

```
1.  iex(1)> import_file "kv.exs"
2.  {:module, KV,
3.   <<70, 79, 82, 49, 0, 0, 7, 16, 66, 69, 65, 77, 69, 120, 68, 99, 0, 0, 0, 99,
    131, 104, 2, 100, 0, 14, 101, 108, 105, 120, 105, 114, 95, 100, 111, 99, 115,
    95, 118, 49, 108, 0, 0, 0, 2, 104, 2, ...>>,
4.   {:loop, 1}}
5.  iex(2)> pid = KV.start_link
6.  #PID<0.73.0>
7.  iex(3)> send pid, {:get, :a, self}
8.  {:get, :a, #PID<0.62.0>}
9.  iex(4)> flush
10. nil
11. :ok
```

Of course, since the map is initially empty, our use of the `:get` message returns `nil`. But, at least it doesn't crash the KV store. Let's add something and attempt to retrieve it again:

```
iex(5)> send pid, {:put, :a, 42, self}
{:put, :a, 42, #PID<0.60.0>}
iex(6)> send pid, {:get, :a, self}
{:get, :a, #PID<0.60.0>}
iex(7)> flush
:ok
42
:ok
```

Remember that the final `:ok` in `flush/0` is the return value of `flush/0`, and not part of the messages returned from the message queue.

After inserting a key into the KV store, we are able to retrieve it as well.

This is an essential pattern for state in processes. We have some map structure and the process will store data. Modifications to this structure are persisted by passing the new version into the loop.

Naming processes

After a while of using process IDs, the requirement to have the PID reference becomes overbearingly tedious. Thus, there is a simpler mechanism for referencing a process if desired—process registration. Instead of referring to the process by the PID reference object, we can register an atom to use for the process ID.

For example, using our key-value store from the previous section, we can register the PID for the process and then refer to it by the atom for all the messages. This is accomplished with the `Process.register/2` function:

```
iex(1)> import_file "kv.exs"
...
iex(2)> pid = KV.start_link
#PID<0.62.0>
iex(3)> Process.register(pid, :kv)
true
```

Now, we can use `send/2` just the same, but instead of passing `pid`, we will pass `:kv`:

```
iex(4)> send :kv, {:put, :a, 42, self}
{:put, :a, 42, #PID<0.60.0>}
iex(5)> flush
:ok
:ok
iex(6)> send :kv, {:get, :a, self}
{:get, :a, #PID<0.60.0>}
iex(7)> flush
42
:ok
```

The code here is the same as before; we are just using the `:kv` process name to reference the running KV store instead of the raw process ID.

Process module

We have used a few functions already from the `Process` module,

namely, `Process.alive?/1` and `Process.register/2`. But there are many more, very useful functions in the `Process` module. I highly recommend going through some of the functions in this module.

Course Content

udemy.com/elixir-scalable-and-efficient-application-development/learn/v4/t/lecture/6958324

Ping pong

Let's start with a very basic example where one process sends a `:ping` message to another process. The receiving process will send a `:pong` message in response.

We will start with a module that looks very similar to the module we created for storing state in a process, except that we have no need for state, this module will only listen for the `:ping` messages and return `:pong`:

```
1.  defmodule PingPong do
2.    def start_link do
3.      spawn_link(fn -> loop() end)
4.    end
5.    defp loop do
6.      receive do
7.        {:ping, sender} ->
8.          send sender, {:pong, self}
9.      end
10.     loop
11.   end
12. end
```

We start with the `start_link/0` function that spawns a new process context and kicks off our internal loop. From the loop, we block with the `receive do` expression. Once the process receives a `:ping` message, it sends back the `:pong` message to the caller. Once this is all complete, it recurses into itself, waiting for the next `:ping` message.

We can load this module up into an interactive session and send it a message to try it out:

```
1.  iex(1)> import_file "pingpong.exs"
2.  {:module, PingPong,
3.   <<70, 79, 82, 49, 0, 0, 7, 196, 66, 69, 65, 77, 69, 120, 68, 99, 0, 0, 0, 99,
    131, 104, 2, 100, 0, 14, 101, 108, 105, 120, 105, 114, 95, 100, 111, 99, 115,
    95, 118, 49, 108, 0, 0, 0, 2, 104, 2, ...>>,
4.   {:loop, 0}}
5.  iex(2)> pid = PingPong.start_link
6.  #PID<0.70.0>
7.  iex(3)> send pid, {:ping, self}
8.  {:ping, #PID<0.60.0>
9.  iex(4)> flush
10. {:pong, #PID<0.70.0>}
11. :ok
```

So far, this example is fairly uninteresting. We are only able to interact with it via the interactive prompt and it's not terribly exciting as we have already done a lot of this, for example, this is a simpler version of the `Worker` module from before.

Let's make it more interesting by adding state and making it more like a heartbeat process.

Heartbeats, in terms of distributed computing, are the concept of pinging or monitoring a process or machine. If the machine does not respond in an acceptable interval, the process or machine is considered dead.

We will develop this version of our `HeartMonitor` module in stages. We will start with defining the messages the `HeartMonitor` process should be listening for, for example, we should definitely receive and handle the `:pong` messages. Another message that should certainly be handled is a message for adding, and, related to it, a message for removing monitors. We should also consider a message for peering into the current monitors, so let's also add a message for `:list_monitors`. Related to listing the current monitors, users of the heartbeat monitor may be curious of what processes are alive and similarly what processes are dead, so we can add patterns for those messages as well. That certainly should be enough messages for a simple heartbeat monitor.

So, we have the following for our internal loop:

```
1.  defp loop(state) do
2.    receive do
3.      {:pong, sender} ->
4.        loop(handle_pong(sender, state))
5.      {:monitor, pid} ->
6.        loop(%{state | :monitors => [pid] ++ state.monitors})
7.      {:demonitor, pid} ->
8.        loop(%{state | :monitors => state.monitors -- [pid]})
9.      {:list_monitors, sender} ->
10.       send sender, {:reply, state.monitors}
11.       loop(state)
12.     {:list_alive, sender} ->
13.       send sender, {:reply, state.alive}
14.       loop(state)
15.     {:list_dead, sender} ->
16.       send sender, {:reply, state.dead}
17.       loop(state)
18.   end
19. end
```

For the `:pong` message, we loop with the result of calling another internal function, `handle_pong`, a function we will get to soon.

When we receive a `:monitor` message, we add the passed `pid` to our internal list of monitors. Similarly, receiving a `:demonitor` message, we remove the given `pid` from our list of monitors.

Finally, all of the `:list_ messages` respond to the calling process with the requested list in a `{:reply, list}` tuple.

Moving to the `handle_pong/2` function, what do we need this function to accomplish? An easy answer is it certainly must return an updated version of the state map since the internal loop expects it. But obviously, it must do more. The heartbeat process will send out pings to each of the monitored processes. In this step, we may create a list of outstanding pings. Thus, the `handle_pong/2` function should resolve the outstanding ping.

Furthermore, we will maintain the list of alive processes; during the handling of a `:pong` message, we need to ensure the responding process is in the alive list. Similarly, it should be removed from the dead list if it was there earlier.

To remove a process from our list of currently dead processes, we will use something similar to the following line of code:

```
1. dead = state.dead -- [sender]
```

If the process is already not a member of the dead list, this set difference will return the dead list, unchanged.

We do a similar operation on the outstanding pings, however. Since the outstanding pings will end up being a map, we can't do the exact same thing:

```
1. pending = Map.delete(state.pending, sender)
```

We will need the list of alive processes locally if we wish to change it:

```
1. alive = state.alive
```

Moreover, if the current responding process isn't in the alive list, we should add it:

```
1. unless sender in state.alive do
2.   alive = [sender] ++ alive
3. end
```

Remember that this reads as "if sender not in alive, then ...".

Finally, we need to update our state map:

```
1. %{state | :alive => alive, :dead => dead, :pending => pending}
```

This maintains the rest of the current state, and we update the lists we (potentially) touched.

Thus, the `handle_pong/2` function in full is:

```
1. defp handle_pong(sender, state) do
2.   dead = state.dead -- [sender]
3.   pending = Map.delete(state.pending, sender)
4.   alive = state.alive
5.   unless sender in state.alive do
6.     alive = [sender] ++ alive
7.   end
8.   %{state | :alive => alive, :dead => dead, :pending => pending}
9. end
```

We are close to having a working, albeit simplistic, heartbeat monitoring process; there are only a few more steps involved.

Next, we need to be able to update our dead list and add processes that haven't responded. An approach to accomplish this would be to check, during heartbeat stage, whether the process has already been added to the outstanding list. If the process is already in the outstanding list, it hasn't responded to the first ping. A more relaxed version

would be to wait until the process is added to the outstanding list for the third time. That is, it hasn't yet responded twice and we are attempting to contact it again. At this point, it may be safe to mark the process as dead. Although this approach is very dependent on the length of time between heartbeat pulses, it should be safe for us to use for now.

We will start by sending pings and updating the outstanding map. This will look something like the following lines of code:

```
1. pending = state.monitors |>
2. Enum.map(fn(p) ->
3.   send p, {:ping, self}
4.   Map.update(state.pending, p, 1, fn(count) -> count + 1 end)
5. end) |>
6. Enum.reduce(%{}, fn(x, acc) ->
7.   Map.merge(x, acc, fn(_, v1, v2) -> v1 + v2 end)
8. end)
```

We map over the current monitors, send a ping to each process, and update or insert the process key into the outstanding or pending map. Since this is a stream, we actually create a new pending list for every monitor. Thus, we must reduce and merge the updated versions. This is accomplished by passing the resulting maps to `Enum.reduce/3` using `Map.merge/3` as the reduction. The merge simply adds the two values together. To convince you that this merge works the way you might expect, let's try it with more simplistic example:

```
1. iex(1)> a = %{:a => 1, :b => 2, :c => 3, :d => 4}
2. %{a: 1, b: 2, c: 3, d: 4}
3. iex(2)> b = %{:a => 4, :b => 3, :c => 2, :d => 1}
4. %{a: 4, b: 3, c: 2, d: 1}
5. iex(3)> [a, b] |> Enum.reduce(%{}, fn(x, acc) ->
6. ...(3)> Map.merge(x, acc, fn(_, v1, v2) -> v1 + v2 end) end)
7. %{a: 5, b: 5, c: 5, d: 5}
```

Here, we create two maps, `a` and `b`. The `a` map has its values increasing from the first key to the last key. And, the `b` map, using the same keys, has its values decreasing from the first key to the last key. The result of merging these two maps should yield a map where the values are all the same.

Next, we need to update the dead process list, based on the result of our new outstanding list. This is accomplished by filtering the processes in the outstanding list that are greater than some threshold, say `2`, and adding it to the dead list:

```
1. dead = (pending |>
2. Enum.filter(fn({p, c}) -> (not p in state.dead) and c > 2 end)
3. Enum.map(fn({p, _}) -> p end)) ++ state.dead
```

This process is more straightforward. The filter will only yield tuples from the pending map that have been inserted more than twice and are not already marked `dead`. The final map strips the count because the dead list is simply a list. Finally, we also want to keep the existing list, so we append the existing list as a final operation.

There are extra parentheses here because of the binding of `++/2`.

Putting together our `send_ping/1` function, we have the following code:

```
1.  defp send_ping(state) do
2.    pending = state.monitors |>
3.    Enum.map(fn(p) ->
4.      send p, {:ping, self}
5.      Map.update(state.pending, p, 1, fn(count) -> count + 1 end)
6.    end) |>
7.    Enum.reduce(%{}, fn(x, acc) ->
8.      Map.merge(x, acc, fn(_, v1, v2) -> v1 + v2 end)
9.    end)
10.   dead = (pending |>
11.   Enum.filter(fn({p, c}) -> (not p in state.dead) and c > 2 end) |>
12.   Enum.map(fn({p, _}) -> p end)) ++ dead
13.   %{state | :pending => pending, :dead => dead}
14. end
```

The final step of the function is to return the new state object, similar to `handle_pong/2`.

Recall the line from `handle_pong/2`, `pending = Map.delete(state.pending, sender)`. We simply remove the entire record of the process. Since, even if we were waiting for it for a while, it responded and thus, it is no longer dead. Note that this is different from healthy, however.

The final step in putting this process together is actually sending the heartbeats. From the perspective of the message blocking interpreting loop, we have nowhere to put this step as part of the process loop. The loop blocks, waiting for a message, possibly receive a message, respond, and recurse. It's possible that the blocking step is the most expensive part. We should do something other than block, for example, send pings.

It turns out that this is a common problem with `receive do` in general. What is a `receive` loop supposed to do if it never receives a message? Thus, there is another clause for `receive` that allows us to do something else if the `receive` clause hangs for too long. This looks similar to the following code:

```
1.  receive do
2.    _ -> 42
3.    after 5000 ->
4.      -42
5.  end
```

If we enter this same expression into `iex`, we have actually created a sleep timer:

```
1.  iex(1)> receive do
2.  ...(1)> _ -> 42
3.  ...(1)> after 5000 -> -42
4.  ...(1)> end
5.  -42
```

Notice, once you enter `end`, the prompt hangs for about 5 seconds and then returns `-42`. This is because the process blocked, waiting for a message, but failed to receive one after 5 seconds, thus executing the expression for the `after` clause.

We can use this exact concept for our heartbeat process. We will add this step to the end of our `loop/1` function:

```
1. {:list_dead, sender} ->
2.   send sender, {:reply, state.dead}
3.   loop(state)
4. after 3000 ->
5.   loop(send_ping(state))
6. end
```

At the end of the message patterns, we insert an `after` clause that will execute our `send_ping/1` after the process fails to receive a message for 3 seconds.

Of course, we will need to add a `start_link/0` function to kick off the whole process:

```
1. def start_link do
2.   spawn_link(fn ->
3.     loop(%{:monitors => [], :alive => [], :dead => [], :pending => %{}})
4.   end)
5. end
```

This is also straightforward; we spawn a link, seeding the default state map of the process.

So the entire `HeartMonitor` module should look similar to the following code:

```
defmodule HeartMonitor do
  def start_link do
    spawn_link(fn ->
      loop(%{:monitors => [], :alive => [], :dead => [], :pending => %{}})
    end)
  end
  defp loop(state) do
    receive do
      {:pong, sender} ->
        loop(handle_pong(sender, state))
      {:monitor, pid} ->
        loop(%{state | :monitors => [pid] ++ state.monitors})
      {:demonitor, pid} ->
        loop(%{state | :monitors => state.monitors -- [pid]})
      {:list_monitors, sender} ->
        send sender, {:reply, state.monitors}
        loop(state)
      {:list_alive, sender} ->
        send sender, {:reply, state.alive}
        loop(state)
      {:list_dead, sender} ->
        send sender, {:relpy, state.dead}
        loop(state)
      after 3000 ->
        loop(send_ping(state))
    end
  end
  defp send_ping(state) do
    pending = state.monitors |>
    Enum.map(fn(p) ->
      send p, {:ping, self}
      Map.update(state.pending, p, 1, fn(count) -> count + 1 end)
    end) |>
    Enum.reduce(%{}, fn(x, acc) ->
      Map.merge(x, acc, fn(_, v1, v2) -> v1 + v2 end)
    end)
    dead = (pending |>
    Enum.filter(fn({p, c}) -> (not p in state.dead) and c > 2 end) |>
    Enum.map(fn({p, _}) -> p end)) ++ state.dead
    %{state | :pending => pending, :dead => dead}
  end
  defp handle_pong(sender, state) do
    dead = state.dead -- [sender]
    pending = Map.delete(state.pending, sender)
    if sender in state.dead do
      IO.puts("Process #{inspect sender} was dead but is now alive")
    end
    alive = state.alive
    unless sender in state.alive do
      alive = [sender] ++ alive
    end
    %{state | :alive => alive, :dead => dead, :pending => pending}
  end
end
```

Simplistically, we should be able to load this up into an interactive session and try it out:

```
iex(1)> import_file "heartmont.exs"
{:module, HeartMonitor,
 <<70, 79, 82, 49, 0, 0, 27, 72, 66, 69, 65, 77, 69, 120, 68, 99, 0, 0, 0, 99,
131, 104, 2, 100, 0, 14, 101, 108, 105, 120, 105, 114, 95, 100, 111, 99, 115,
95, 118, 49, 108, 0, 0, 0, 2, 104, 2, ...>>,
 {:handle_pong, 2}}
iex(2)> heart_pid = HeartMonitor.start_link
#PID<0.67.0>
iex(3)> Process.alive?(heart)
true
```

Now that the heartbeat process is alive, we can begin monitoring processes and watching the alive process list change. For example, we can start by monitoring the REPL process:

```
iex(3)> send heart_pid, {:monitor, self}
{:monitor, #PID<0.60.0>}
```

You should begin to see messages pile up into the REPL inbox:

```
iex(4)> flush
{:ping, #PID<0.67.0>}
{:ping, #PID<0.67.0>}
{:ping, #PID<0.67.0>}
{:ping, #PID<0.67.0>}
{:ping, #PID<0.67.0>}
{:ping, #PID<0.67.0>}
{:ping, #PID<0.67.0>}
:ok
```

We can also request to see which processes are dead:

```
iex(5)> send heart_pid, {:list_dead, self}
{:list_dead, #PID<0.60.0>}
iex(6)> flush
{:ping, #PID<0.67.0>}
{:ping, #PID<0.67.0>}
{:ping, #PID<0.67.0>}
{:reply, [#PID<0.60.0>]}
:ok
```

We are still receiving pings because the REPL process is still being monitored. We could, at this point, also send back a `:pong` message:

```
iex(7)> send heart_pid, {:pong, self}
Process #PID<0.60.0> was dead but is now alive
{:pong, #PID<0.60.0>}
```

As far as the heartbeat process is concerned, the REPL process is alive again because it is responding to the pings. However, after 9 seconds, that will no longer be the case.

Let's create a separate process by which we can have the heartbeat process monitor.

If you are going to keep your existing `iex` open, be sure to demonitor the REPL: `send heart_pid, {:demonitor, self}`.

The new process will be similar to the worker, but it will be specific to the heartbeat process. That is, it must know how to respond to the `:ping` messages. However, all the other messages are free:

```
defmodule NewWorker do
  def start do
    spawn(fn -> loop() end)
  end
  defp loop do
    receive do
      {:ping, sender} ->
        send sender, {:pong, self}
        loop()
      {:compute, n, sender} ->
        send sender, {:result, fib(n)}
        loop()
    end
  end
  defp fib(0), do: 0
  defp fib(1), do: 1
  defp fib(n), do: fib(n-1) + fib(n-2)
end
```

Our worker here accepts two messages, `:ping` and `:compute`. The `:ping` message is as we expect and the `:compute` message requests the computation of the n###sup/sup### digit of the Fibonacci sequence.

Let's hook it into the heartbeat process:

```
iex(1)> import_file "heartmon.exs"
...
iex(2)> import_file "newworker.exs"
...
iex(3)> heart_pid = HeartMonitor.start_link
#PID<0.130.0>
iex(4)> w1 = NewWorker.start
#PID<0.137.0>
```

After importing the two modules and starting the processes for each, we can add the worker to the monitoring process:

```
iex(5)> send heart_pid, {:monitor, w1}
{:monitor, #PID<0.137.0>}
```

We should see the process now in the alive list:

```
iex(6)> send heart_pid, {:list_alive, self}
{:monitor, #PID<0.123.0>}
iex(7)> flush
{:reply, [#PID<0.137.0]}
:ok
```

Next, we can send compute messages to the worker to see it perform some work for us, and we can see that the monitoring process doesn't really mind:

```
iex(8)> send w1, {:compute, 5, self}
{:compute, 5, #PID<0.123.0>}
iex(9)> send w1, {:compute, 20, self}
{:compute, 20, #PID<0.123.0>}
iex(10)> flush
{:reply, 5}
{:reply, 6765}
:ok
```

Asking for alive processes again should still show the worker process:

```
iex(11)> send heart_pid, {:list_alive, self}
{:list_alive, #PID<0.123.0>}
iex(12)> flush
{:reply, [#PID<0.137.0>]}
:ok
```

Now, let's try having the worker compute out the sequence a bit:

```
iex(13)> send w1, {:compute, 40, self}
{:compute, 40, #PID<0.123.0>}
iex(14)> flush
:ok
...
Process #PID<0.137.0> was dead but is now alive
```

It takes a while to compute the 40###sup/sup### digit of the Fibonacci sequence, occasionally longer than 9 seconds. Thus, we see a message from the heart monitor process about the process being marked `dead` and then coming back to life later:

```
iex(15)> send heart_pid, {:list_alive, self}
{:list_alive, #PID<0.123.0>}
iex(16)> flush
{:reply, [#PID<0.137.0>]}
:ok
```

Once the worker has finished and stabilized, we should see it now listed in the alive list.

As a sort of disclaimer, this heartbeat monitoring process we have just developed is very incomplete and will likely not do well for production use, or, really, any serious use. Furthermore, it has a number of performance problems that will cause the process to no longer work, especially if the number of monitored processes exceeds a relatively small number.

Course Content

The queue portion arises from the need to still schedule work while the pool is fully consumed.

For the next example, let's go through the process of developing a work pool and job queue.

Typical of a work pool is to have some sort of scheduling process. For simplicity, we can use a FIFO-scheduler. It will simply compute work as it arrives; there is no special (re)ordering.

For this example, we will create it inside a project instead of a single file. Let's start by creating the project:

```
$ mix new workpool
* creating README.md
* creating .gitignore
* creating mix.exs
* creating config
* creating config/config.exs
* creating lib
* creating lib/workpool.ex
* creating test
* creating test/test_helper.exs
* creating test/workpool_test.exs
Your mix project was created successfully.
You can use mix to compile it, test it, and more:
    cd workpool
    mix test
Run `mix help` for more commands.
```

We will then start by creating the scheduler process inside the `./lib/workpool/` folder.

Creating a folder with the same name as the root module is standard practice when creating submodules, for example, `Workpool.Scheduler`, of the root or base module.

So create the directory and let's start into creating the scheduler:

```
$ mkdir lib/workpool
$ touch lib/workpool/schduler.ex
```

We will create our typical internal loop that performs message handling. We will respond to several messages such as work queuing and the `:DOWN` messages to notify requesters of failure.

This version of the work pool will be unbounded; it will accept and perform as much work as it is given. We will later introduce an artificial limit and create some back pressure to not collapse the system.

For the internal loop, we do what we might expect. Wait and respond to messages with a `receive do` loop:

```
defp loop(state) do
  receive do
    {:queue, func, args, sender} ->
      {pid, _ref} = spawn_monitor(fn ->
        send sender, func.(args)
      end)
      processing = Dict.put(state.processing, pid, sender)
      loop(%{state | processing: processing})
    {:DOWN, _ref, :process, pid, :normal} ->
      loop(%{state | processing: Dict.delete(state.processing, pid)})
    {:DOWN, _ref, :process, pid, reason} ->
      sender = Dict.get(state.processing, pid)
      send sender, {:failure, "pool worker died: #{inspect reason}"}
      loop(%{state | processing: Dict.delete(state.processing, pid)})
  end
end
```

When the scheduler receives a `:queue` message, it spawns a worker process that will apply the given function over the given arguments. The worker will also attempt to send the results back to the sender as is. Since the scheduler will have a monitor to the worker process, the scheduler is notified when the worker finishes or exits for a non-normal reason. In case the worker exits for normal reasons, the scheduler will simply remove the process from the processing dictionary; otherwise, the scheduler should notify the calling process about the failure and then remove it from the processing queue.

The `start_link` is the same as usual:

```
def start_link do
  spawn_link(fn -> loop(%{processing: HashDict.new()}) end)
end
```

Thus, the whole `Workpool.Scheduler` module consists of the following code:

```
defmodule Workpool.Scheduler do
  def start_link do
    spawn_link(fn -> loop(%{processing: HashDict.new()}) end)
  end
  defp loop(state) do
    receive do
{:queue, func, args, sender} ->
        {pid, _ref} = spawn_monitor(fn ->
          send sender, func.(args)
        end)
        processing = Dict.put(state.processing, pid, sender)
        loop(%{state | processing: processing})
      {:DOWN, _ref, :process, pid, :normal} ->
        loop(%{state | processing: Dict.delete(state.processing, pid)})
      {:DOWN, _ref, :process, pid, reason} ->
        sender = Dict.get(state.processing, pid)
        send sender, {:failure, "pool worker died: #{inspect reason}"}
        loop(%{state | processing: Dict.delete(state.processing, pid)})
    end
  end
end
```

We can then compile and load the project into an interactive session:

```
$ iex -S mix
Erlang/OTP 18 [erts-7.0] [source] [64-bit] [smp:12:12] [async-threads:10]
[hipe] [kernel-poll:false]
Compiled lib/workpool.ex
Compiled lib/workpool/scheduler.ex
Generated workpool app
Interactive Elixir (1.0.5) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)>
```

From the `iex` prompt, we can start up the scheduler and begin passing work to it:

```
iex(1)> scheduler = Workpool.Scheduler.start_link
#PID<0.101.0>
iex(2)> send scheduler, {:queue, fn(x) -> 2 * x * x end, 4, self}
{:queue, #Function<6.54118792/1 in :erl_eval.expr/5>, 4, #PID<0.99.0>}
iex(3)> flush
32
:ok
```

So far, it seems it is working as we would expect.

As we are using monitors instead of links, the scheduler shouldn't fail if an exception is raised inside a worker, child process:

```
iex(4)> send scheduler, {:queue, fn(_) -> raise "oops" end, [], self}
01:34:58.604 [error] Process #PID<0.108.0> raised an exception
** (RuntimeError) oops
    (workpool) lib/workpool/scheduler.ex:11: anonymous fn/3 in
Workpool.Scheduler.loop/1
{:queue, #Function<6.54118792/1 in :erl_eval.expr/5>, [], #PID<0.99.0>}
iex(5)> Process.alive?(scheduler)
true
```

And as we would expect, the worker process fails and the scheduler process is still alive.

This workpool scheduler is somewhat tedious to use. The consuming process has to know a few things about the scheduler, for example, its process ID. We can use process registration to help, but how would the consuming process ensure it's only starting a single scheduler or registering over an existing process?

A way this could be solved is to have the `Workpool` module do the start up and process registration. That way, it's in one place and all the consuming processes should be able to simply use the registered atom instead of having to know the PID reference.

To accomplish this, we need to add a function to the `Workpool` module, say, `start/0` s:

```
1. def start do
2.   pid = Workpool.Scheduler.start_link
3.   true = Process.register(pid, :scheduler)
4.   :ok
5. end
```

This way, we can launch the scheduler from the `Workpool` module, and the consumers of the work pool do not need to know about the `Workpool.Scheduler` module. Let's see what it looks similar to now to interact with the `Workpool` module:

```
1. $ iex -S mix
2. Erlang/OTP 18 [erts-7.0] [source] [64-bit] [smp:12:12] [async-threads:10]
   [hipe] [kernel-poll:false]
3. Compiled lib/workpool.ex
4. Compiled lib/workpool/scheduler.ex
5. Generated workpool app
6. Interactive Elixir (1.0.5) - press Ctrl+C to exit (type h() ENTER for help)
7. iex(1)> Workpool.start
8. :ok
```

Now that the scheduler process is started, we should be able to send messages to it using the registered name:

```
1. iex(2)> send :scheduler, {:queue, fn(x) -> x * x end, 7, self}
2. {:queue, #Function<6.54118792/1 in :erl_eval.expr/5>, 7, #PID<0.84.0>}
3. iex(3)> flush
4. 49
5. :ok
```

There is something else we should add to make this work pool better. Sending messages between processes can often be done incorrectly. There is so much variability in the tuple that is passed, and there is no good way for the sending process to know it made a mistake.

We can add some functions to the `Workpool` module that can be the public API for the scheduler. Then, consuming processes of the work pool don't need to know the message format, and they are more easily made aware of changes to the API. Since we really only have a single public message, we only need to add a `queue/2` function:

```
def queue(fun, args) do
  send :scheduler, {:queue, fun, args, self}
  :ok
end
```

This way, we can interact with the `Workpool` module via simply `start/0` and `queue/2`.

Here is the new final version of `Workpool`:

```
defmodule Workpool do
  def start do
    pid = Workpool.Scheduler.start_link
    true = Process.register(pid, :scheduler)
    :ok
  end
  def queue(fun, args) do
    send :scheduler, {:queue, fun, args, self}
    :ok
  end
end
```

Let's recompile and see this in action:

```
$ iex -S mix
...
iex(1)> Workpool.start
:ok
iex(2)> Workpool.queue(fn(x) -> x * x * 2 end, 4)
:ok
iex(3)> flush
32
:ok
```

This will be much easier to use for other end consumers. They will simply need to familiarize themselves with the `Workpool` module's API, and they would be set. Let's create a very basic module that uses the work pool and show, indeed, how easy it is now to use the work pool.

This won't be part of the work pool project, but we will want to be able to load both the work pool project and this new module into the same `iex` session.

Let's create a module that computes a number of Fibonacci numbers, but uses the work pool to do this in parallel:

```
defmodule FibonacciWorkPool do
  def fib(fibonacci_digits) do
    fibonacci_digits |>
    Enum.map(fn(n) -> Workpool.queue(&fib_comp/1, n) end)
  end
  # Terribly slow version
  defp fib_comp(0), do: 0
  defp fib_comp(1), do: 1
  defp fib_comp(n), do: fib_comp(n-1) + fib_comp(n-2)
end
```

The main function is the `fib/1` function that takes an enumerable type and attempts to map it into the workpool scheduler using a very slow method of finding the Fibonacci number.

Let's load up the work pool, import this module, and see it in action:

```
1. $ iex -S mix
2. ...
3. iex(1)> Workpool.start
4. :ok
```

We need the work pool going, but this could come after importing the `FibonacciWorkPool` module:

```
1. iex(2)> import_file "../fib_pool.exs"
2. {:module, FibonacciWorkPool,
3.  <<70, 79, 82, 49, 0, 0, 6, 172, 66, 69, 65, 77, 69, 120, 68, 99, 0, 0, 0, 125,
   131, 104, 2, 100, 0, 14, 101, 108, 105, 120, 105, 114, 95, 100, 111, 99, 115,
   95, 118, 49, 108, 0, 0, 0, 2, 104, 2, ...>>,
4.  {:fib_comp, 1}}
```

Now that the module is loaded, we can start asking it for multiple Fibonacci numbers, all to be delivered to the `iex` message queue:

```
iex(3)> FibonacciWorkPool.fib(5..35)
[:ok, :ok, :ok, :ok, :ok, :ok, :ok, :ok, :ok, :ok, :ok, :ok, :ok, :ok, :ok, :ok,
 :ok, :ok, :ok, :ok, :ok, :ok, :ok, :ok, :ok, :ok, :ok, :ok, :ok, :ok, :ok]
iex(4)> flush
5
8
13
21
34
89
55
144
233
377
610
987
2584
4181
1597
10946
17711
6765
46368
28657
75025
121393
196418
317811
514229
832040
1346269
2178309
3524578
5702887
9227465
:ok
```

Of course, small digits of the Fibonacci sequence are going to return almost immediately. Let's try with some bigger numbers:

```
iex(5)> FibonacciWorkPool.fib(39..45)
[:ok, :ok, :ok, :ok, :ok, :ok, :ok]
```

You should see your CPU start to cycle up for all of the separate computations. I waited about 10 to 30 seconds for the CPU to cycle down before proceeding:

```
iex(6)> flush
63245986
102334155
165580141
267914296
433494437
701408733
1134903170
:ok
```

The output being sent to a mailbox isn't ideal, but could be handled by more sophisticated work pool clients. But the use of the work pool was really only one line. The Fibonacci module had only to know of the `Workpool.queue/2` function to function. Using the Fibonacci module without starting the work pool would be a disaster. But under the current assumptions, it's not worth fixing.

The work pool is a great example of server-worker scheduling method that can be greatly expanded to solve real problems; however, in its current form, we have to deal with the Erlang/Elixir process API too much before we even get to a stable point. There should be a better way to implement everything we have so far and then some without a lot of extra effort.

Summary

There's a lot of interesting things we can do with Erlang/Elixir processes and the concurrency they enable. However, there are a lot of disadvantages of using them in their raw form. As you might have noticed, we have duplicated a lot of work throughout the examples. To summarize, we covered the following topics:

- We wrote a lot of similar functions for spawning the processes, the receive loops, message delivery failure, and so on.
- We saw that monitoring and managing processes usually results in some duplication of effort.
- The work pool example started into a form of process supervision, but it's missing a few cases and is not entirely robust.
- Similarly, we also saw there's a lot of variability with the order and parent processes that start other processes, which can lead to false assumptions and failing invariants that would ultimately crash the system as a whole.

Quiz Time!

Quiz 7 | 2 Questions

Start Quiz Skip Quiz >

Question 1:

Process monitors are...

- Stackable links
- Non-stackable links
- Directional links
- Mutable links

Question 2:

How does the process modify Elixir immutable data?

- ○ Recursive finite loop
- ○ While loop
- ○ Tail-recursive infinite loop
- ○ For loop