Supervisors

SimAlchemy, Part 3

Have you tried
turning it off and back on?

Why is this a thing?!

- Computers can get stuck in bad cycles
- One common cause for this is reaching unexpected internal states

Why does it sometimes work???

- Often going back through startup returns things to a known good state
- It’s essentially a reset switch for state
- Of course, it doesn’t always work

More Process Primitives

The case for spawn_link()

Monitors

- A monitor is a one-way relationship
- The monitoring process is sent messages when the monitored process stops
- This is ideal for resource cleanup
- Start a monitored process: spawn_monitor()
- Monitor an existing process: Process.monitor(pid_or_name)

Monitor Example

```
spawn_monitor(fn -> IO.puts "#{inspect self} exiting…" end)

receive do
  notification ->
    IO.puts "#{inspect self} received:  #{inspect notification}"
end

#PID<0.73.0> exiting…
#PID<0.70.0> received:  {:DOWN, #Reference<0.0.6.88>, :process, #PID<0.73.0>, :normal}
```

Links

- Links are two-way
- If a linked process crashes, all processes it is linked to also crash
- Links are the glue of supervision trees
- Start a linked process: spawn_link()
- Link to an existing process: Process.link(pid_or_name)

Link Example

```
{pid, reference} = spawn_monitor(fn ->
  spawn_link(fn -> IO.puts "#{inspect self} crashing…"; raise "Oops" end)
  receive do :ping -> IO.puts "Pong" end
end)

receive do
  {:DOWN, ^reference, :process, ^pid, _reason} -> IO.puts "#{inspect pid} exited."
end

#PID<0.74.0> crashing…
#PID<0.73.0> exited.
# 17:37:29.617 [error] Process #PID<0.74.0> raised an exception
# ** (RuntimeError) Oops
#     link.exs:2: anonymous fn/0 in :elixir_compiler_0.__FILE__/1
```

Trapping Exits

- A linked process can choose not the die with the link
- Instead they will receive a message about the process that went down
- This is how supervisors do their thing
- Start trapping exits: Process.flag(:trap_exit, true)

Trapping Exits Example

```
{pid, reference} = spawn_monitor(fn ->
  Process.flag(:trap_exit, true)
  link = spawn_link(fn -> IO.puts "#{inspect self} crashing…"; raise "Oops" end)
  receive do {:EXIT, ^link, _reason} -> IO.puts "#{inspect link} crashed." end
  receive do :ping -> IO.puts "Pong" end
end)

send(pid, :ping)
receive do
  {:DOWN, ^reference, :process, ^pid, :normal} -> IO.puts "#{inspect pid} exited."
end

#PID<0.74.0> crashing…
#PID<0.74.0> crashed.
#Pong
#PID<0.73.0> exited.
```

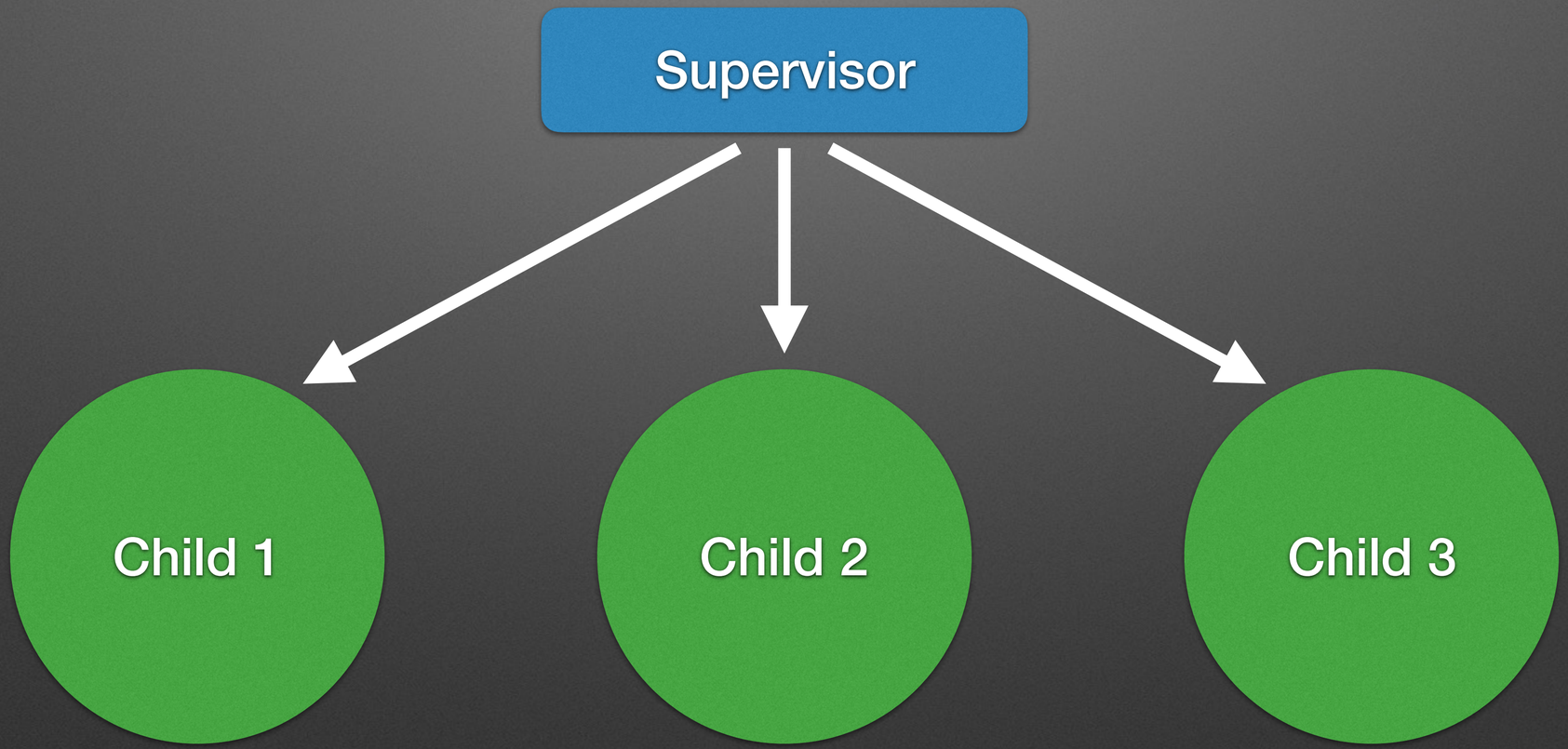
Meet OTP's Supervisor

It turns processes off and back on for you

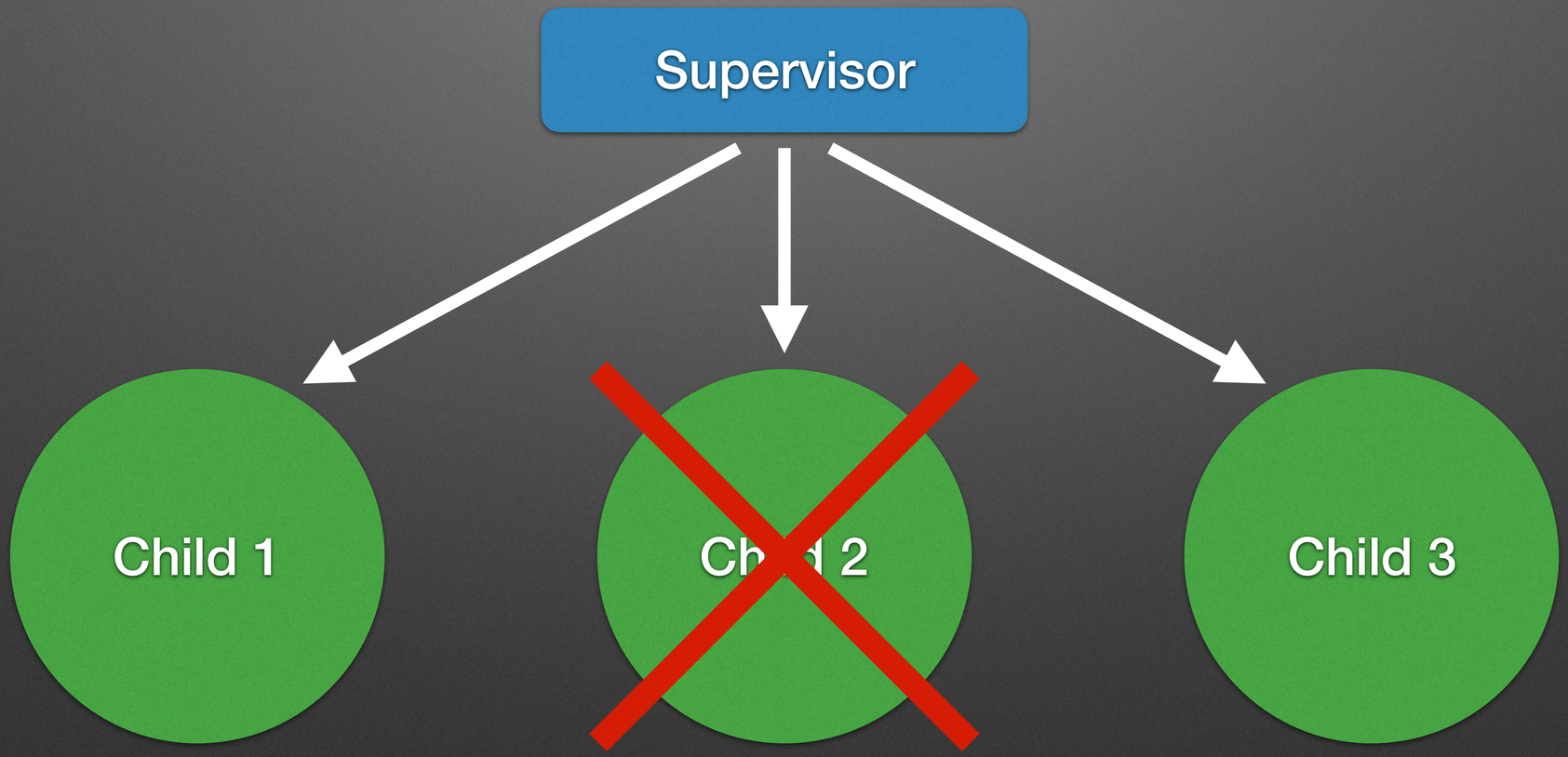
The Big Idea

- All processes are started under the control of a supervisor
- If a process dies, the supervisor tries to restart it
- Supervisors have some configuration:
  - Which processes they try to restart
  - How often they try to restart before they give up

Strategies

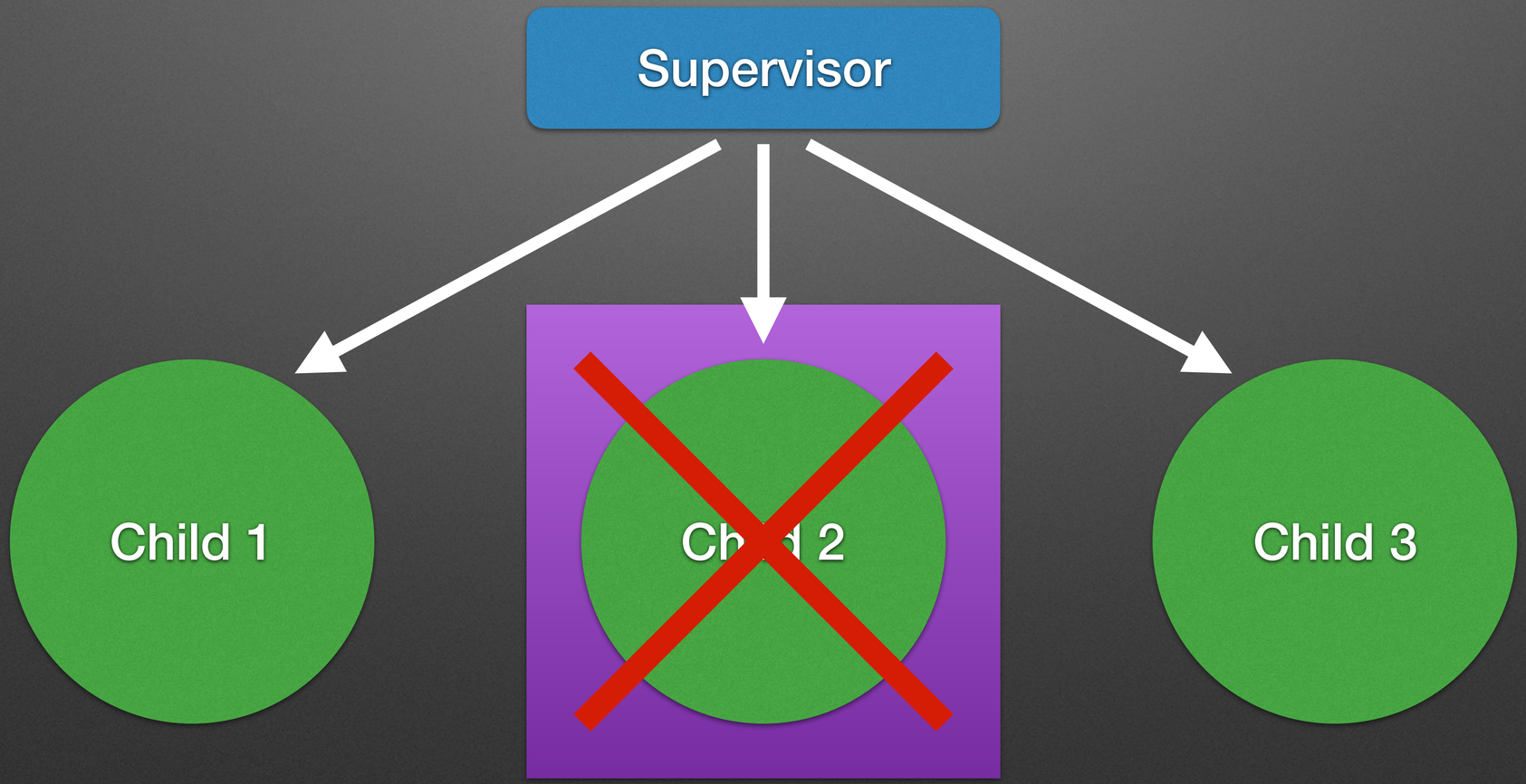


Strategies



Strategies

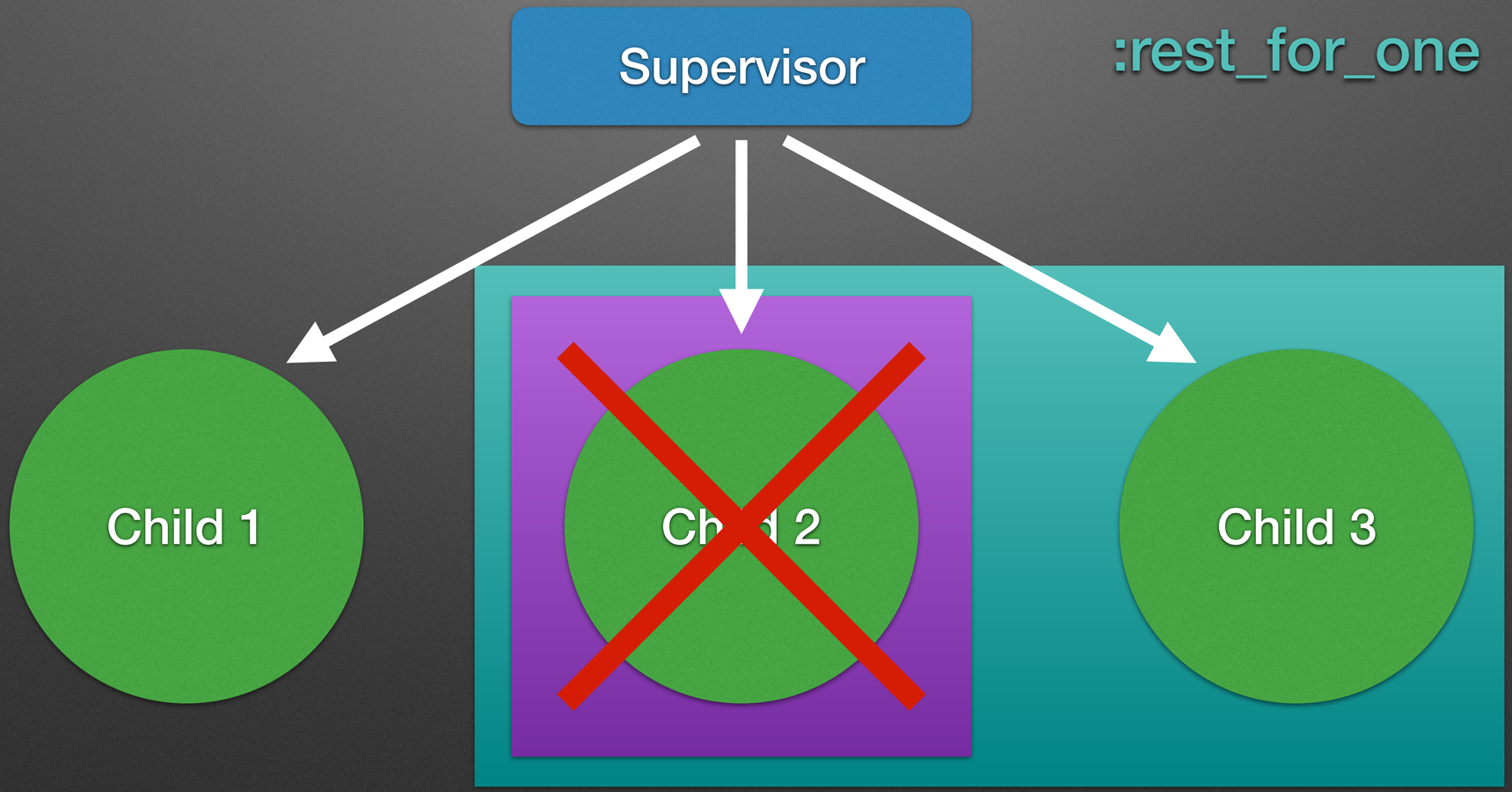
:one_for_one



Strategies

:one_for_one

:rest_for_one

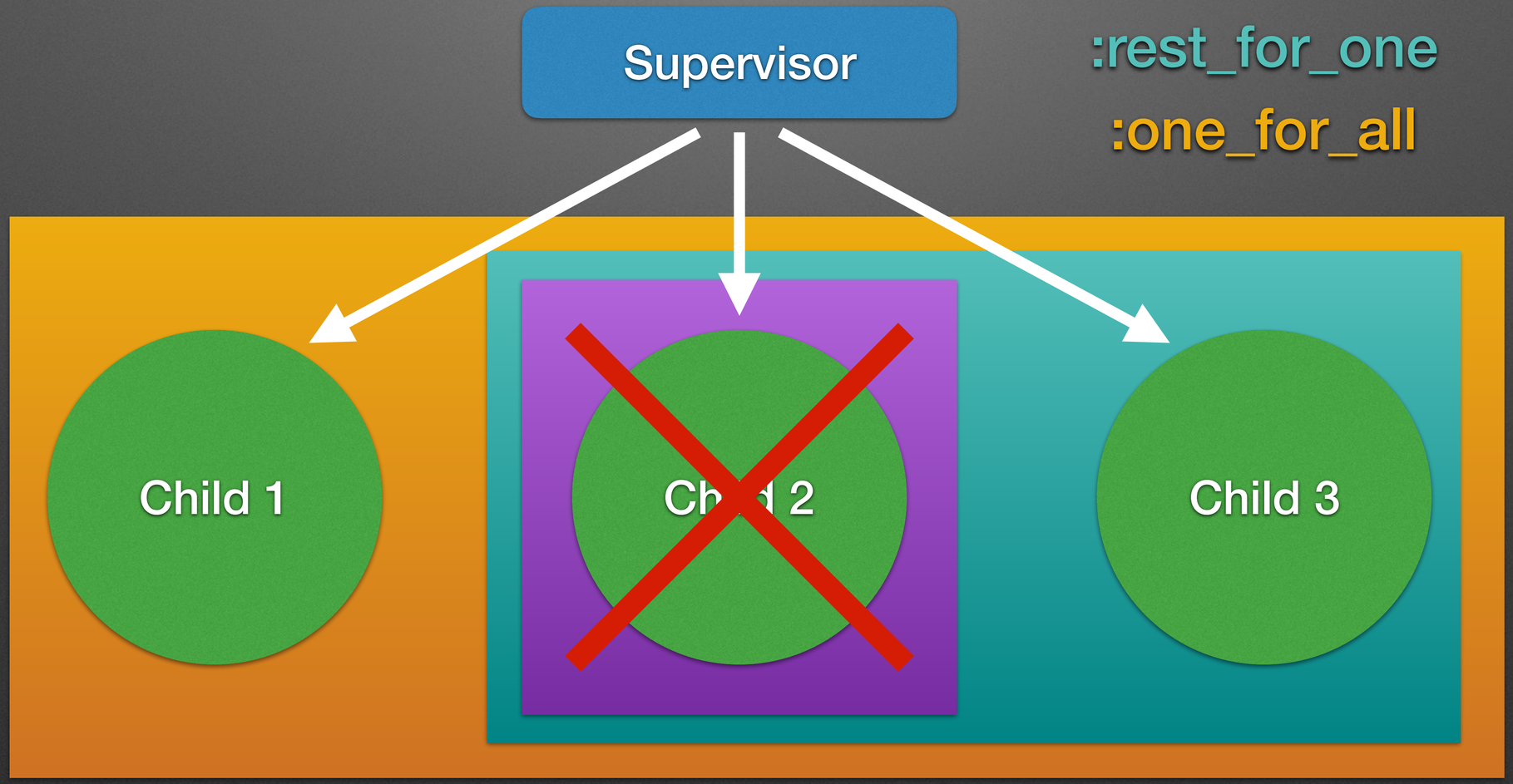


Strategies

:one_for_one

:rest_for_one

:one_for_all



Restart Limits

- :max_restarts allowed in…
- :max_seconds
- Example: max_restarts: 3, max_seconds: 5
  - Allow up to 3 restarts in a 5 second window
  - Give up if this is exceeded
- No exponential backoff built-in

“Let it Crash!”

- A philosophy about how supervisors should influence our designs
- We shouldn’t add a ton of error handling code for every imaginable scenario
  - It’s OK to let our process crash in such states
  - The supervisor will replace it

“Let it Crash!” Example

```
defmodule LetItCrash do
  use GenServer

  def start_link, do: GenServer.start_link(__MODULE__, nil)
  def supported(pid), do: GenServer.call(pid, :supported)

  def handle_call(:supported, _from, nil), do: {:reply, :ok, nil}
  # NOT:  def handle_call(_request, _from, nil), do: {:reply, :ok, nil}
end

{:ok, pid} = LetItCrash.start_link
LetItCrash.supported(pid)           # :ok
GenServer.call(pid, :unsupported)   # (FunctionClauseError) … LetItCrash.handle_call/3
```

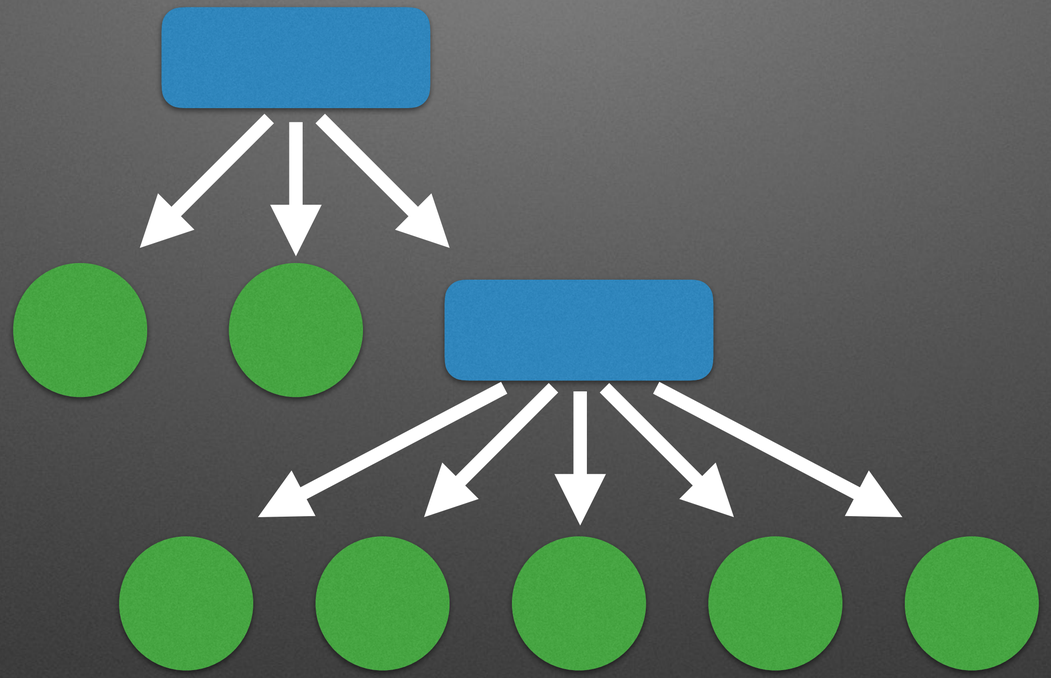
Not For Everything

- Supervisors would hammer an API returning 404 if you “Let it Crash!”
- Supervisors are for **unexpected errors**
- You still need to handle expected errors
  - 400 HTTP status codes are a good example of your responsibility

Supervision Trees

- Supervisors can supervise other supervisors
- This allows us to build a tree of process
  - Each branch can have different supervision configuration
- This also supports things like clean shutdown
  - We tell the supervisor at the top and let it trickle down

A Sample Tree



Giving Up

- If a supervisor can’t keep a process up, it will give up and crash
- This causes larger portions of the tree to be restarted
- More state is reset by this rising tide
- If we keep failing at the top, your program will crash to indicate the problem

The Code

How we create supervisors

Two Ways to Make Supervisors

- You can generally use the shortcut interface: Supervisor.Spec
- More complex needs require a module with callbacks (like GenServer)

mix new —sup my_project

```
defmodule MyProject do
  use Application

  # See http://elixir-lang.org/docs/stable/elixir/Application.html
  # for more information on OTP Applications
  def start(_type, _args) do
    import Supervisor.Spec, warn: false

    # Define workers and child supervisors to be supervised
    children = [
      # Starts a worker by calling: MyProject.Worker.start_link(arg1, arg2, arg3)
      # worker(MyProject.Worker, [arg1, arg2, arg3]),
    ]

    # See http://elixir-lang.org/docs/stable/elixir/Supervisor.html
    # for other strategies and supported options
    opts = [strategy: :one_for_one, name: MyProject.Supervisor]
    Supervisor.start_link(children, opts)
  end
end
```

Supervisor.Spec API

- Build a list of children with:
  - worker(module, args, options \\ [ ])
  - supervisor(module, args, options \\ [ ])
- Then pass that list to:
  - Supervisor.start_link(children, options)

A Module Supervisor

```elixir
defmodule MyProject.Supervisor do
  use Supervisor

  def start_link, do: Supervisor.start_link(__MODULE__, nil, [name: __MODULE__])

  def init(nil) do
    # special setup code goes here…

    children = [
      # worker(MyProject.Worker, [arg1, arg2, arg3]),
    ]
    supervise(children, strategy: :one_for_one)
  end
end
```

Supervisor Module Differences

- use Supervisor (instead of Supervisor.Spec)
- Call Supervisor.start_link() with module and args, instead of children
- Provide init() callback
- The last line of init() should be supervise(children, options)

Dynamic Supervisors

:simple_one_for_one

- A supervision strategy that supports starting/stoping new children as needed
- You give the supervisor one “template”
- Then you can start new instances on demand
- Arguments from the template and Supervisor.start_child() are combined

Worker :restart Options

- When should the supervisor restart a process?
  - :permanent means always
  - :temporary means never
  - :transient means if it dies abnormally, but not if it exits cleanly
    - This is often combined with :simple_one_for_one

A Dynamic Supervisor Example

```
defmodule Worker do
  use GenServer

  def start_link(work, modifier) do
    GenServer.start_link(__MODULE__, {work, modifier})
  end
  def init(state) do
    send(self, :work)
    {:ok, state}
  end

  def handle_info(:work, state = {work, modifier}) do
    work |> Enum.map(fn n -> n * modifier end) |> Enum.join(", ") |> IO.puts
    {:noreply, state}
  end
end
```

A Dynamic Supervisor Example

```
defmodule DynamicSupervisor do
  def start_link do
    import Supervisor.Spec, warn: false
    children = [worker(Worker, [1..10], [restart: :transient])]
    Supervisor.start_link(children, strategy: :simple_one_for_one)
  end
end

{:ok, supervisor} = DynamicSupervisor.start_link
Supervisor.start_child(supervisor, [2])  # prints:  2, 4, 6, 8, 10, 12, 14, 16, 18, 20
Supervisor.start_child(supervisor, [3])  # prints:  3, 6, 9, 12, 15, 18, 21, 24, 27, 30
Process.sleep(1_000)
```

Process Registration

A tradeoff of supervision

Where is my process?

- With spawn_link() you get a PID and send() to that PID
- When supervisors launch processes for you, you need to abandon PIDs
  - A process could be replaced by the supervisor making your PID useless
- Supervisors are not directory services
  - Trying to look your processes up in their child lists is the wrong fix

Process Registration

- When a process starts, you can register it
- Later, you can lookup that process by its current registration
- For simple cases: name the process
- For distributed cases: use :global
- Most other cases: use gproc

Named Processes Example

```elixir
defmodule NamedProcess do
  use GenServer

  def start_link, do: GenServer.start_link(__MODULE__, nil, [name:  __MODULE__])
  def greet, do: GenServer.call(__MODULE__, :greet)

  def handle_call(:greet, _from, nil) do
    {:reply, "Hello from #{__MODULE__}!", nil}
  end
end

NamedProcess.start_link
NamedProcess.greet  # => "Hello from Elixir.NamedProcess!"
```

gproc

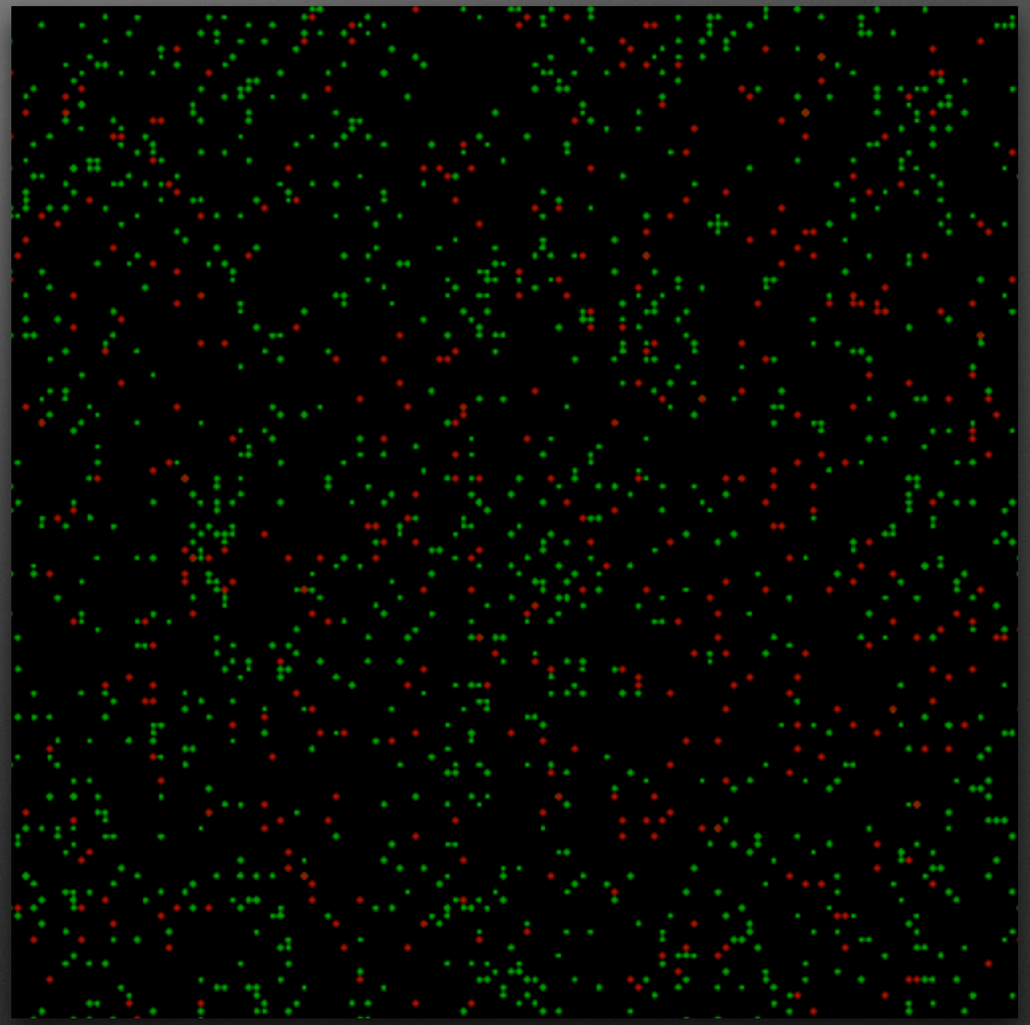
- An Erlang dependency you can get from Hex
- Allows you to name a process using arbitrary terms
  - GenServer.start_link(…, name: {:via, :gproc, {:n, :l, {:turtle, 42}}})
  - GenServer.call({:via, :gproc, {:n, :l, {:turtle, 42}}}, …)
- Also supports grouping processes and Pub/Sub

Turtle Ecology Supervisors Simulation

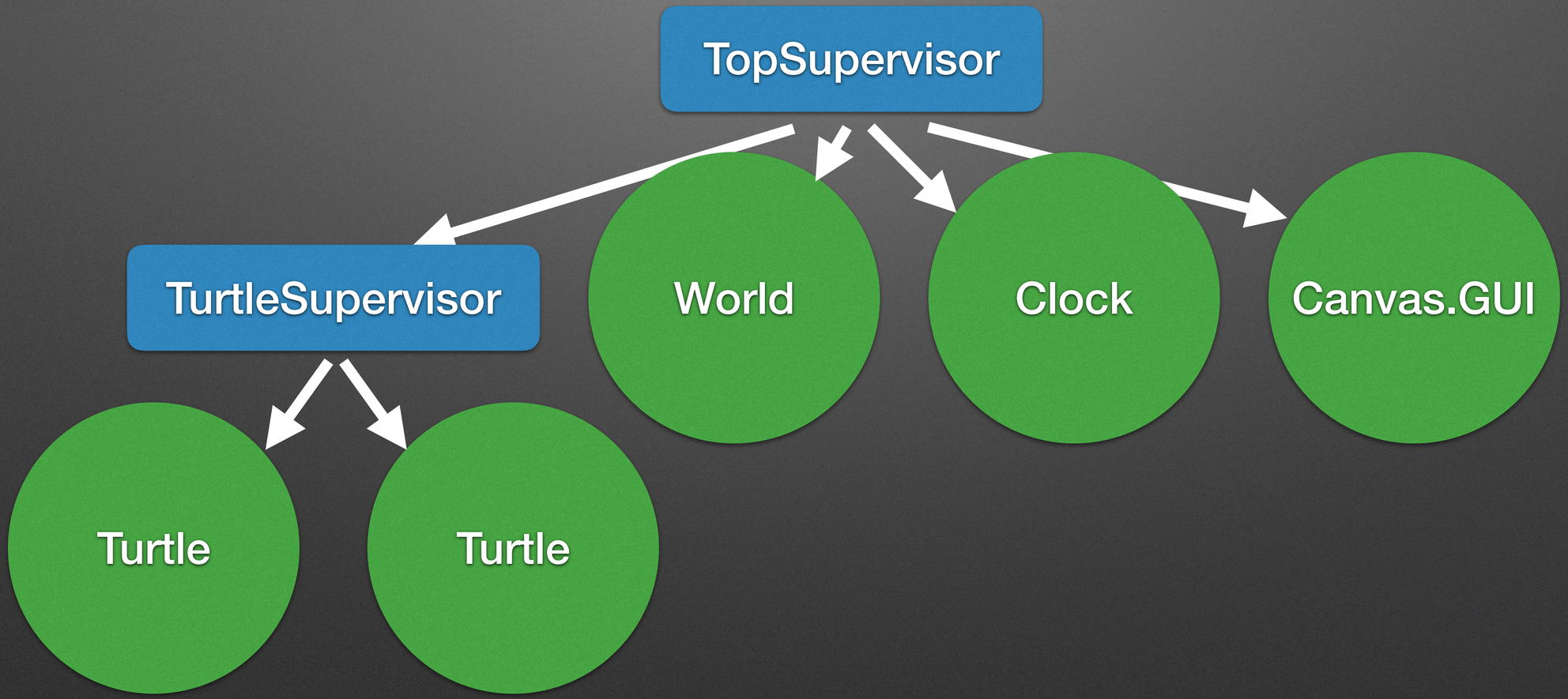
Exercise 3

Building a Supervision Tree

- One dynamic Supervisor manages Turtle processes
- Another Supervisor manages:
  - The Turtle Supervisor
  - The World process
  - The Clock process
  - A Canvas.GUI process that does the drawing



The Supervision Tree



Hints

- Turtle.start/2 needs to return the top-level Supervisor of the tree
- Everything is provided, expect the two supervisors
- You can use the Supervisor.Spec shortcut syntax to define both
- Remember that a Supervisor will call a module’s start_link() function (look at those for the arguments you need to pass)

http://bit.ly/tesupsimzip

See instructions in README.md