GenServer and Friends

SimAlchemy, Part 2

What do we want out of our processes?

Goals

- Robustness
- Performance
- Discoverability
- Support for common usage patterns, like replies
- Easy debugging

OTP Rules

- How a process should be started
- Hooks processes need to support
  - System messages, like shutdown
  - Debugging messages

GenServer

The default OTP process

Adds More Rules

- Follows OTP Rules
- Takes over the inner process event loop
- Specifies how messages should be sent and received

Downsides

- More complex than basic processes
- Abstracts simple tools, like send() and receive do … end

Wins: Our Goals

- Robustness
  - Support supervision
- Discoverability
  - Support process registration
- Performance
  - Support synchronous and asynchronous messaging

Wins: Common Patterns

- GenServer abstracts process usage
- Is built around client/server architecture
- Can wait for replies
- Applies timeouts
- Provides hooks for upgrades and cleanup

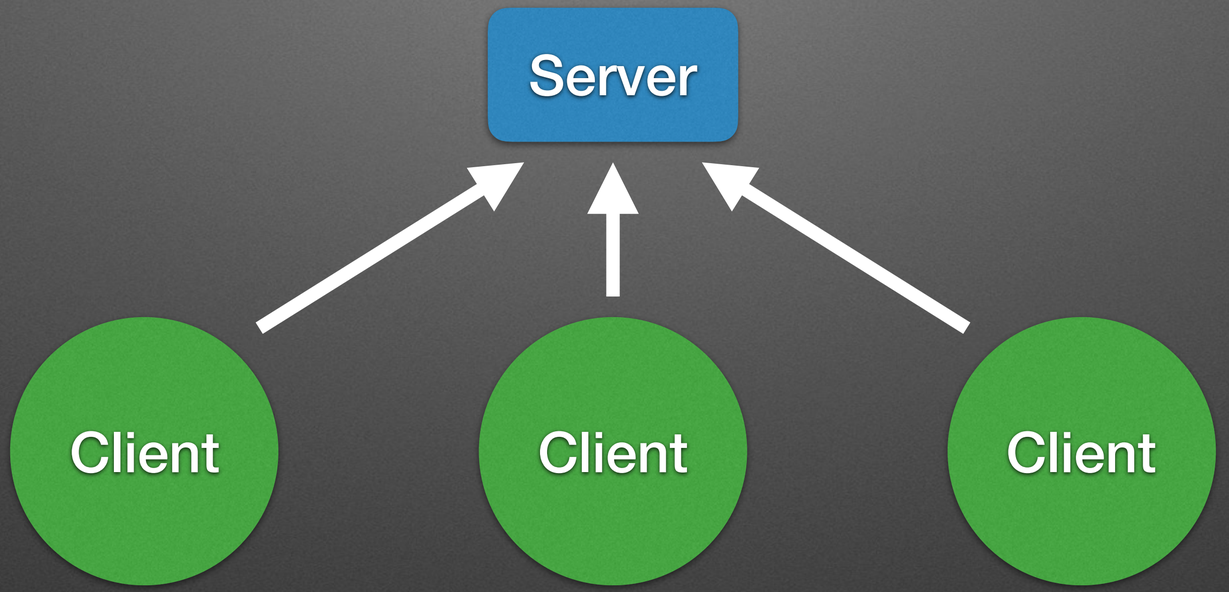
Wins: Debugging

- Makes it possible to introspect processes
- Supports tracing events

Callbacks
(The Server Interface)

- init(args)
- handle_call(request, from, state)
- handle_cast(request, state)
- handle_info(message, state)
- terminate(reason, state)
- code_change(old_vsn, state, extra)

The Client/Server Model



The Same Module Approach

```
defmodule MyProcess do
  use GenServer

  # Client API functions first:  wrapping GenServer.*(…)

  # Server Callbacks next:  init(…), handle_*(…), etc.
end
```

init(args)

- Called when a process starts
- Should setup state or cancel the start
- Common return values:
  - {:ok, state}
  - {:stop, reason}

handle_call(request, from, state)

- When a client sends a request and waits for an answer
- The primary GenServer communication tool
- Common return values
  - {:reply, reply, new_state}
  - {:stop, reason, reply, new_state}

handle_cast(request, state)

- When a client sends a message and doesn’t wait
- Still common, but can be dangerous
- Common return values:
  - {:noreply, new_state}
  - {:stop, reason, new_state}

Call or Cast?

- If you need an answer, use call
- Even if you don’t, consider call and replying with :ok
- Cast can lead to great performance
  - But you can get buried in work
- It’s generally best favor call, until you have clear use cases for cast

handle_info(message, state)

- When you receive a normal (non-call or cast) message
- Consider a catch-all that logs unexpected messages
- Handling a lot of messages this way is probably a smell
- Common return values: same as cast

terminate(reason, state)

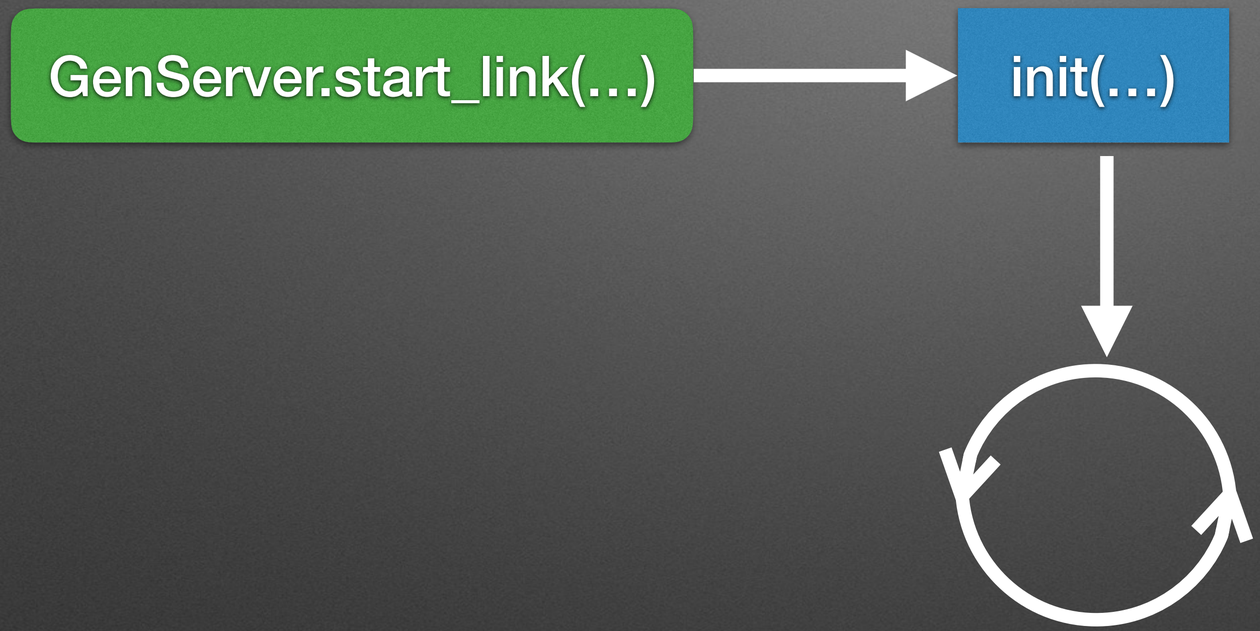
- Called when a process is stopping or being shutdown
- A chance to do cleanup
- Not called if your process is killed
  - Move critical cleanup to another process

The Client Interface

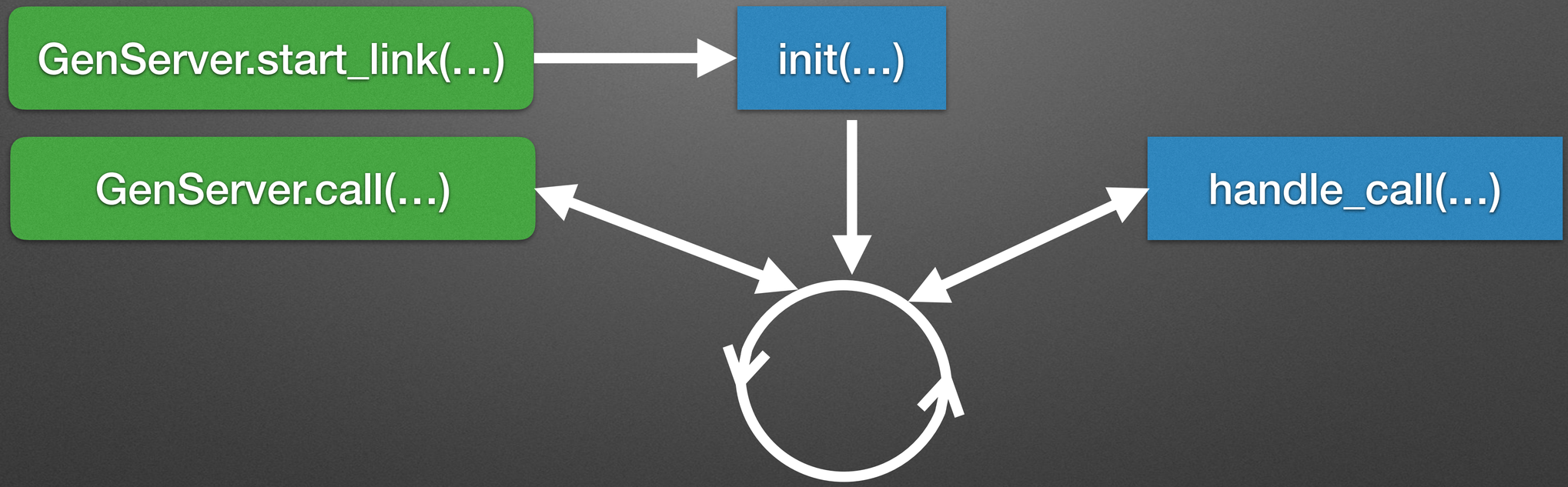
- pid = GenServer.start_link(module, args, options \\ [ ])
- reply = GenServer.call(server, request, timeout \\ 5000)
- GenServer.cast(server, request)
- GenServer.stop(server, reason \\ :normal, timeout \\ :infinity)

The GenServer Flow

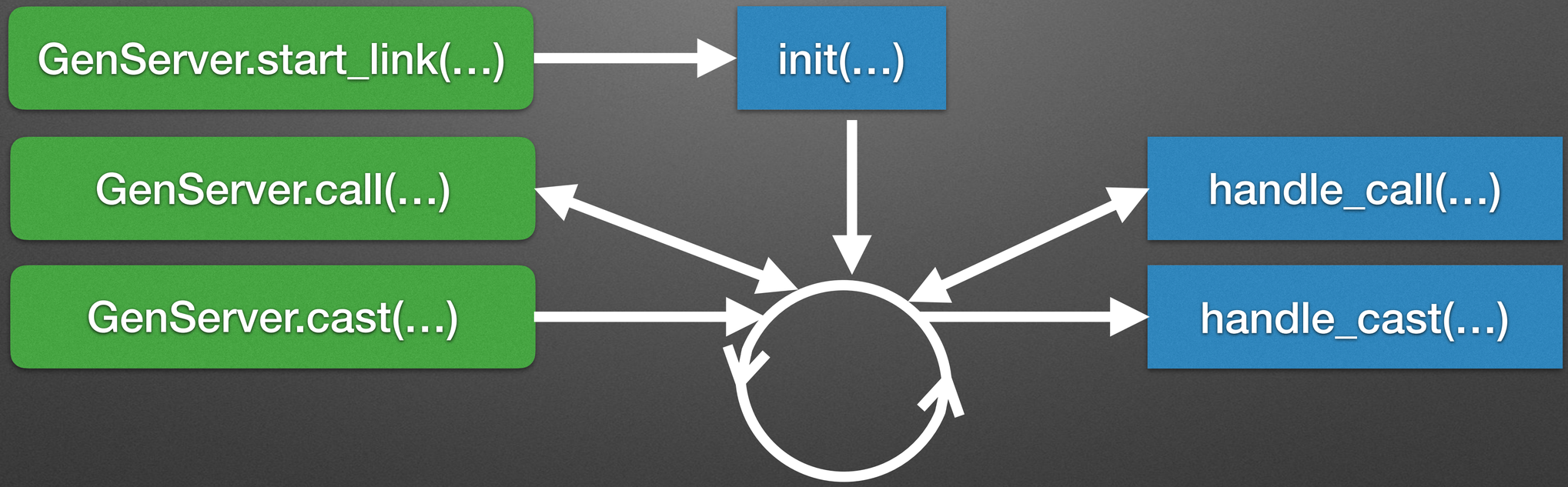
The GenServer Flow



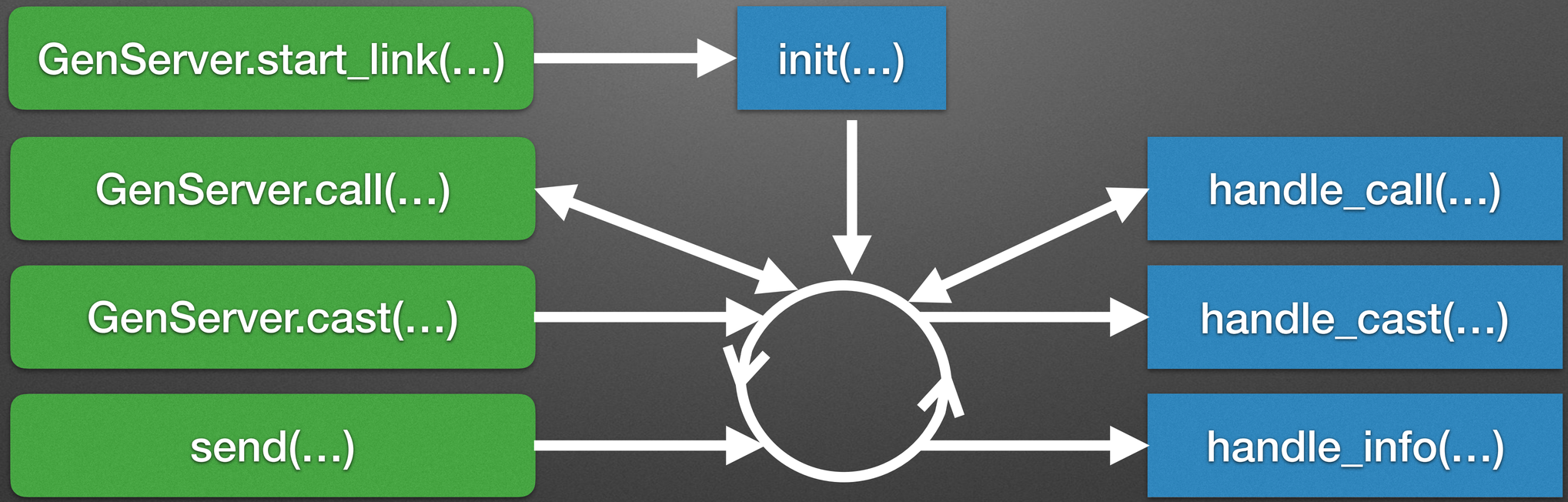
The GenServer Flow



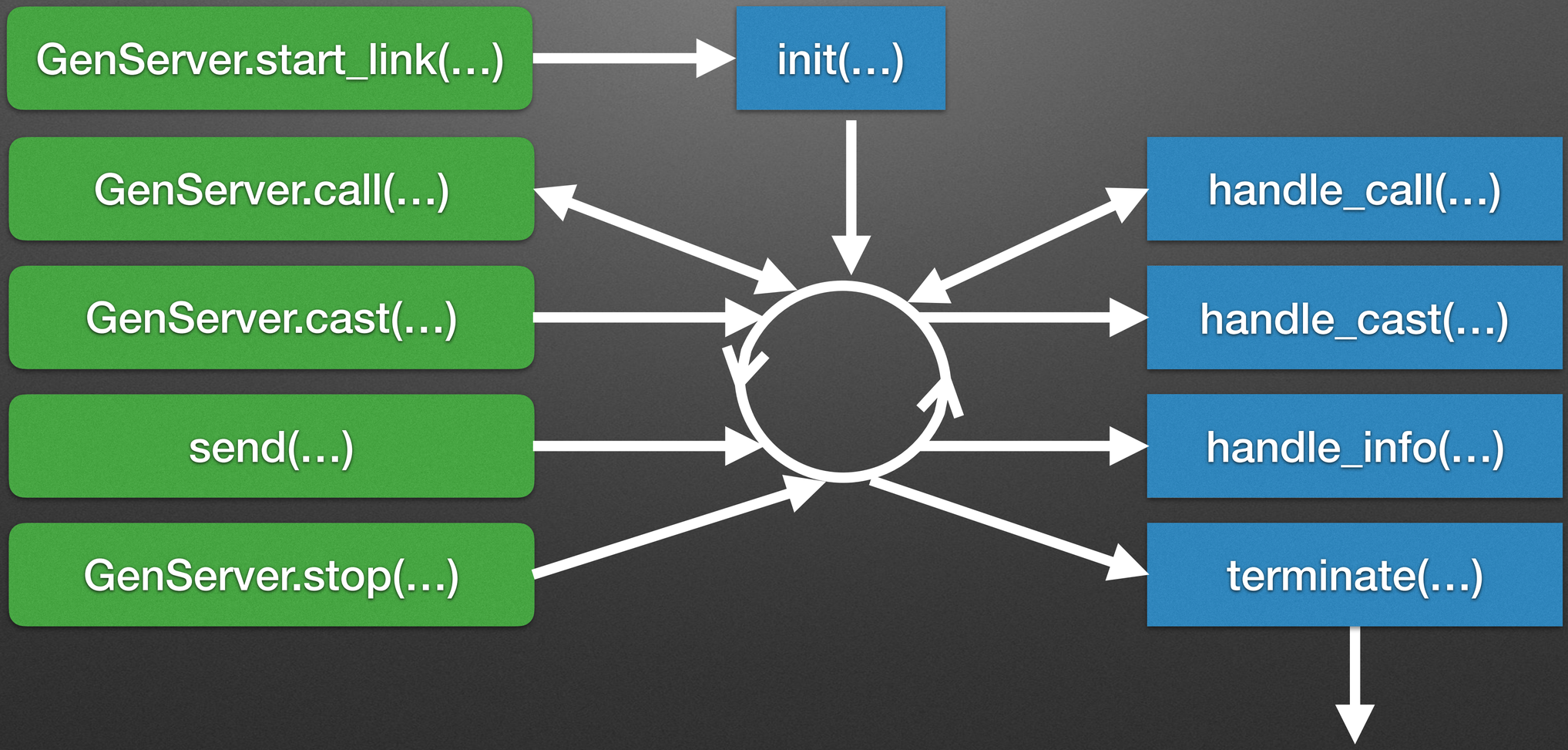
The GenServer Flow



The GenServer Flow



The GenServer Flow



GenServer Example

```
defmodule SalesTracker do
  use GenServer

  def start_link, do: GenServer.start_link(__MODULE__, [ ])
  def record(tracker, sales), do: GenServer.call(tracker, {:record, sales})

  def init([ ]), do: {:ok, {elem(:calendar.local_time, 0), 0}}
  def handle_call({:record, new_sales}, _from, {previous_date, sales}) do
    date = elem(:calendar.local_time, 0)
    if date == previous_date do
      {:reply, :ok, {date, sales + new_sales}}
    else
      IO.puts "#{inspect previous_date}:  #{sales}"
      {:reply, :ok, {date, new_sales}}
    end
  end
end
```

GenServer Example

```
{:ok, tracker} = SalesTracker.start_link
SalesTracker.record(tracker, 21)
SalesTracker.record(tracker, 21)
# ... the next day ...
SalesTracker.record(tracker, 7)  # prints:  {2016, 8, 19}:  42
```

Other OTP Processes

From Erlang

- gen_event
  - wrapped by Elixir’s GenEvent
  - likely to lose favor to GenStage
- gen_fsm
  - likely to lose favor to gen_statem

Elixir’s Additions

- Agent
  - Good for storing, manipulating, and reading data
- Task
  - Good for asynchronous work
  - Good for processes that don’t communicate
  - Can be dynamically supervised with Task.Supervisor

Agent

- start_link(fun, options \\ [ ])
- update(agent, fun, timeout \\ 5000)
- get(agent, fun, timeout \\ 5000)
- get_and_update(agent, fun, timeout \\ 5000)

Agent Example

```
defmodule ChangeLog do
  def start_link, do: Agent.start_link(fn -> [ ] end)

  def record(agent, change) do
    Agent.update(agent, fn changes -> [change | changes] end)
  end

  def get_changes_and_clear(agent) do
    Agent.get_and_update(agent, fn changes -> {Enum.reverse(changes), [ ]} end)
  end
end

{:ok, log} = ChangeLog.start_link
Enum.each(1..3, fn n -> ChangeLog.record(log, "Change #{n}") end)
ChangeLog.get_changes_and_clear(log)  # => ["Change 1", "Change 2", "Change 3"]
ChangeLog.get_changes_and_clear(log)  # => [ ]
```

Task Example

```
defmodule ParallelEnum do
  def map(enumerable, fun) do
    enumerable
    |> Enum.map(fn item -> Task.async(fn -> fun.(item) end) end)
    |> Enum.map(&Task.await/1)
  end
end

1..10
|> ParallelEnum.map(fn n -> n * n end)
|> IO.inspect
```

“Special Processes”

- The OTP defines the rules to make your own
  - See the documentation for details
- This is rare
  - You typically just use GenServer and friends

Turtle Ecology Simulation

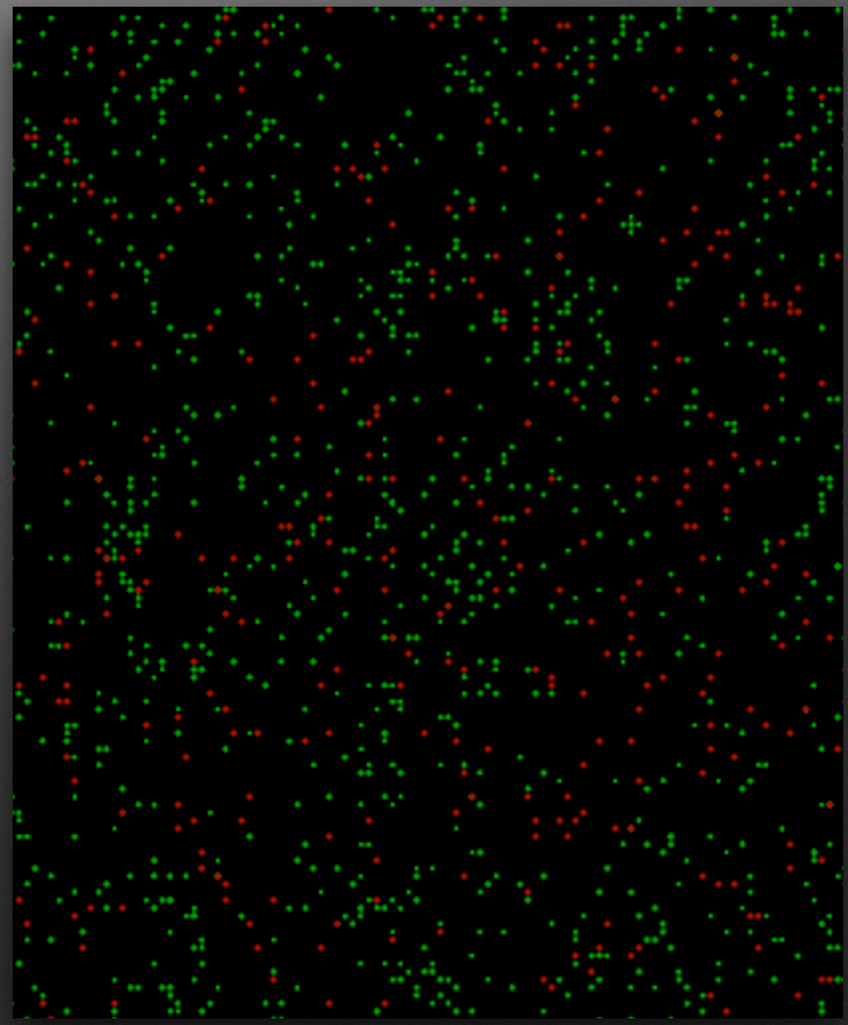
Exercise 2

The Simulation

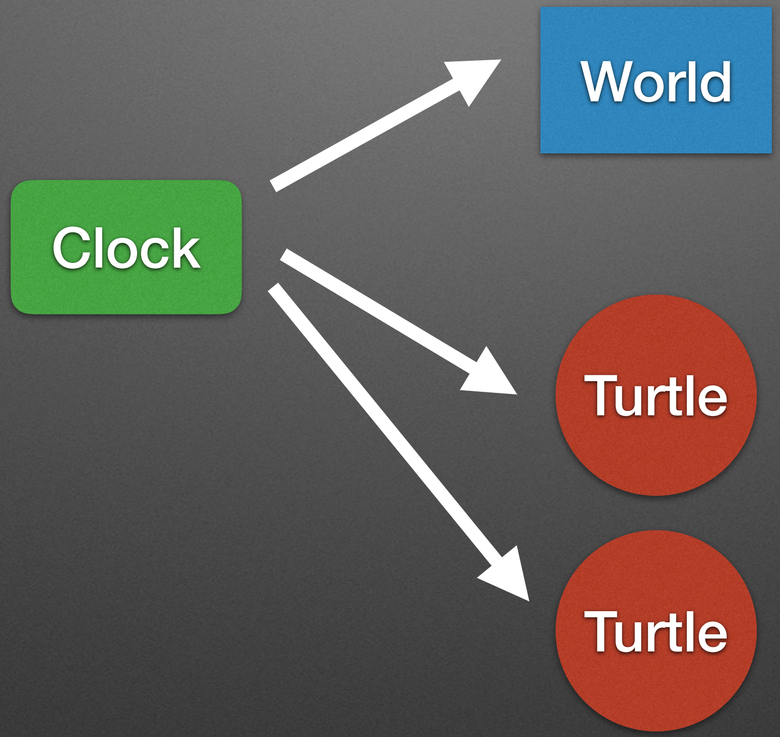
- Turtles eat food when standing on it to gain energy
- Otherwise they spend energy to walk around
- If they collect a surplus of energy, they give birth to a new turtle
- Turtles that run out of energy die
- Food randomly grows from time to time

Building the Clock

- One process is responsible for managing time
  - In the beginning it places plants and turtles in the world
  - As time advances, it tells turtles to act
  - It must track births and deaths, so all live turtles will always receive act messages



The Processes



Hints

- You’re making a GenServer process
- Your Clock is passed the world, the world’s size, and a function that starts new turtles when it launches
- You’ll need three callbacks init/1, handle_call/3, and handle_info/2

More Hints

- :rand.uniform(n) returns a random number between 1 and n (inclusive)
- World has place_plants/2 and place_turtles/2
- Turtle.act/1 returns :ok if there's no new birth or the pid of a newborn Turtle
- Process.monitor(pid) will send your process a {:DOWN, _ref, :process, pid, _reason} message when pid dies

http://bit.ly/tesimzip

See instructions in README.md