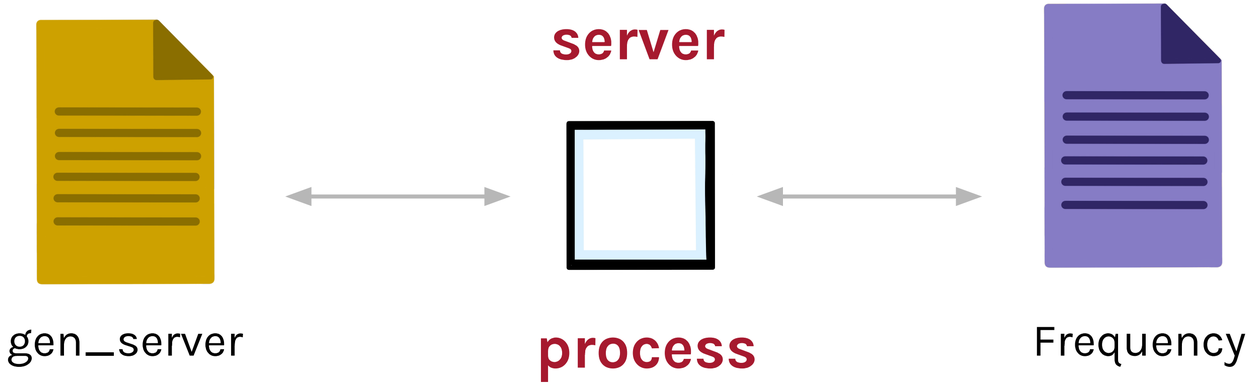
GENERIC SERVERS

Generic Servers

- Generic Servers
- Starting a Server
- Message Passing
- Termination
- Other Messages
- Timeouts
- Other Issues

Generic Servers



- The **gen_server** module implements the client-server behaviour
  - It is part of the standard library application
- The library module contains all the generic code
- Non-generic code is placed in the callback module

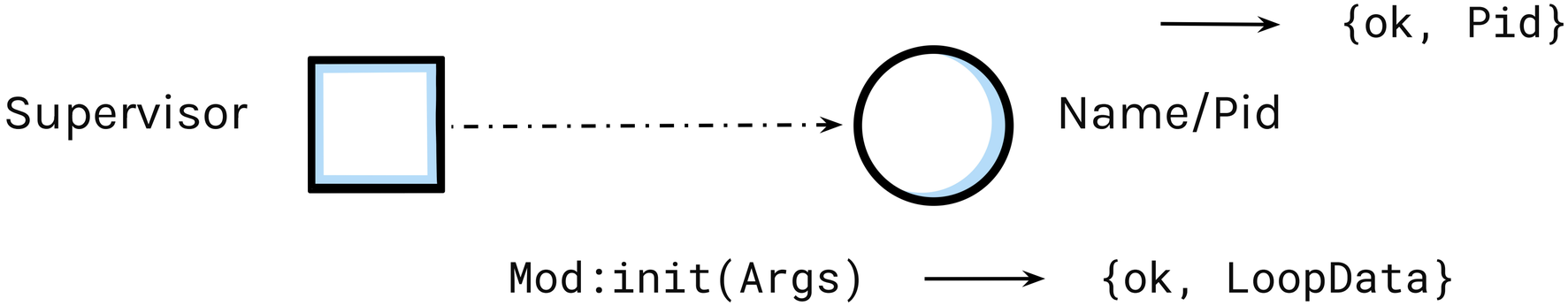
Generic Servers

```
-module(frequency).
-behaviour(gen_server).

-export([start_link/1, init/1, ...]).

start_link(...) -> ...
```

- Call back modules must have one extra directive
- **behaviour** directive, used at compile time

Starting a Server

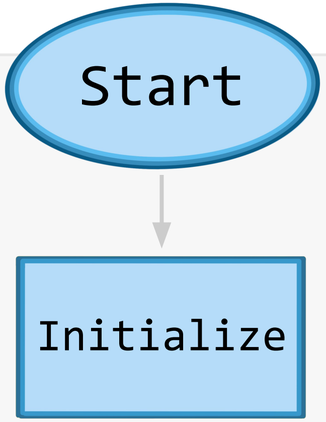
```
gen_server:start_link({local, Name}, Mod, Args, [])
```



- **gen_server:start_link/4** creates a new server
  - Name is the process name
  - Mod is the name of the callback module
  - Args are the arguments passed to the init function
- **init/1** is called by the server callback module
  - It initialises the process state and returns the loop data

Starting a Server

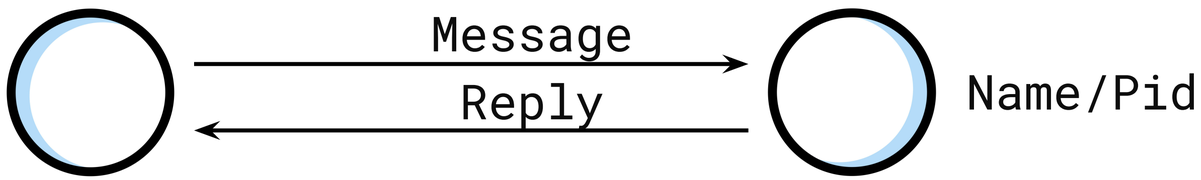
```
-module(frequency).
-behaviour(gen_server).

-export([start_link/0, init/1, ....]).

start_link() ->
 gen_server:start_link({local,frequency},frequency,[],[]).

init(_Args) ->
    Free = get_frequencies(),
    Allocated = [],
    {ok, {Free, Allocated}}.

get_frequencies() -> [10,11,12,13,14,15].
```



server name

Message Passing: synchronous

```
gen_server:call(Name|Pid, Message) ⟶  Reply
```

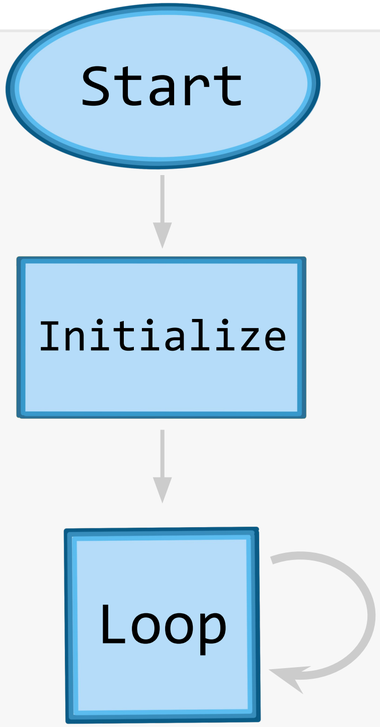



```
Mod:handle_call(Message, _From, LoopData)
                          ↓
{reply, Reply, NewLoopData}
```

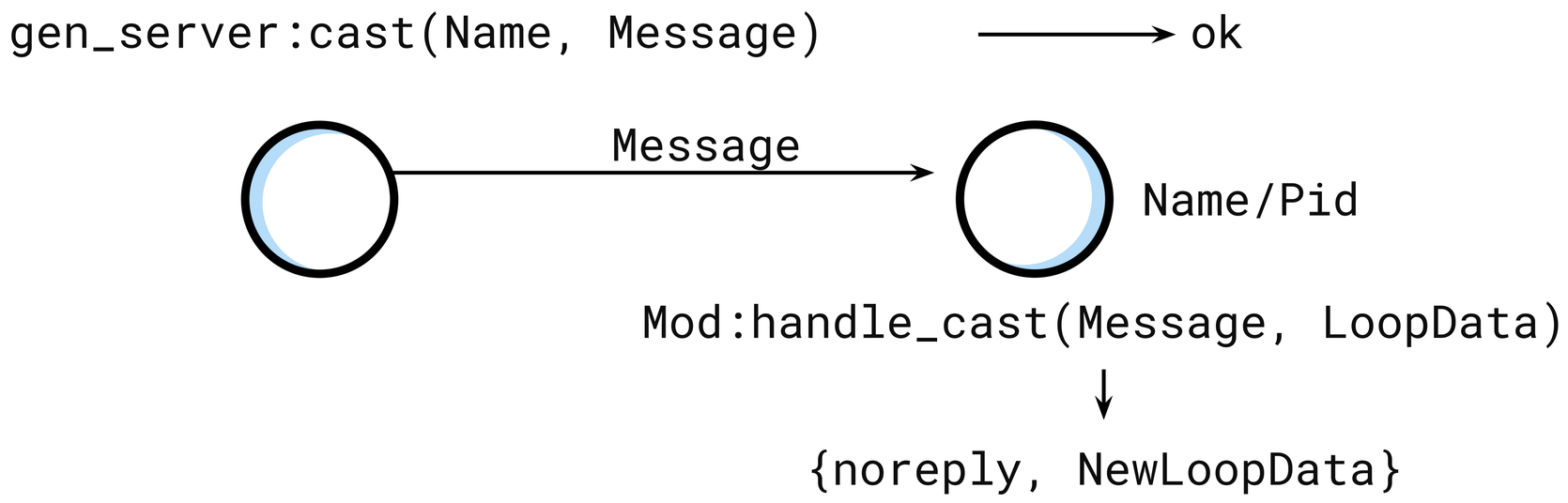
- **gen_server:call/2** is used by the client to send requests
  - Requests are synchronous
  - Handled in the **handle_call/3** function in the callback module
- **handle_call/3** returns the tuple **{reply, Reply, NewLoopData}**.

Message Passing: synchronous

```
init(_Args) ->
    Free = get_frequencies(),
    Allocated = [],
    {ok, {Free, Allocated}}.

get_frequencies() -> [10,11,12,13,14, 15].

allocate() ->
    gen_server:call(frequency, {allocate, self()}).

handle_call({allocate, Pid}, _From, Frequencies) ->
    {NewFrequencies, Reply} = allocate(Frequencies, Pid),
    {reply, Reply, NewFrequencies}.
```



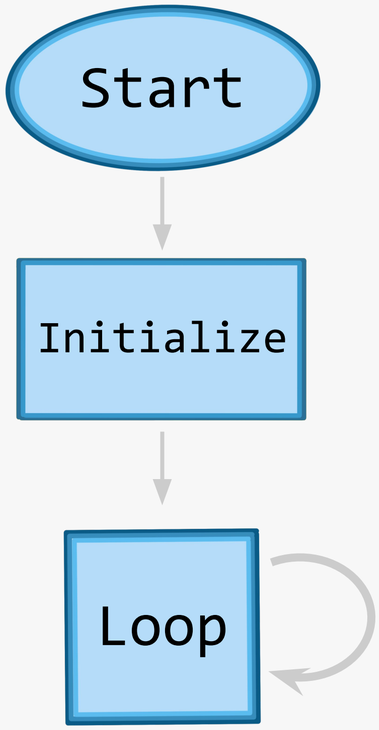
Message Passing: asynchronous



- **gen_server:cast/2** is used by the client to send messages
  - Requests are asynchronous
  - Handled in the **handle_cast/2** function in the callback module
- **handle_cast/2** returns the tuple **{noreply, NewLoopData}**.

Message Passing: asynchronous

```
deallocate(Freq) ->
   gen_server:cast(frequency, {deallocate, Freq}).

handle_cast({deallocate, Freq}, Frequencies) ->
    NewFrequencies = deallocate(Frequencies, Freq),
    {noreply, NewFrequencies}.
```



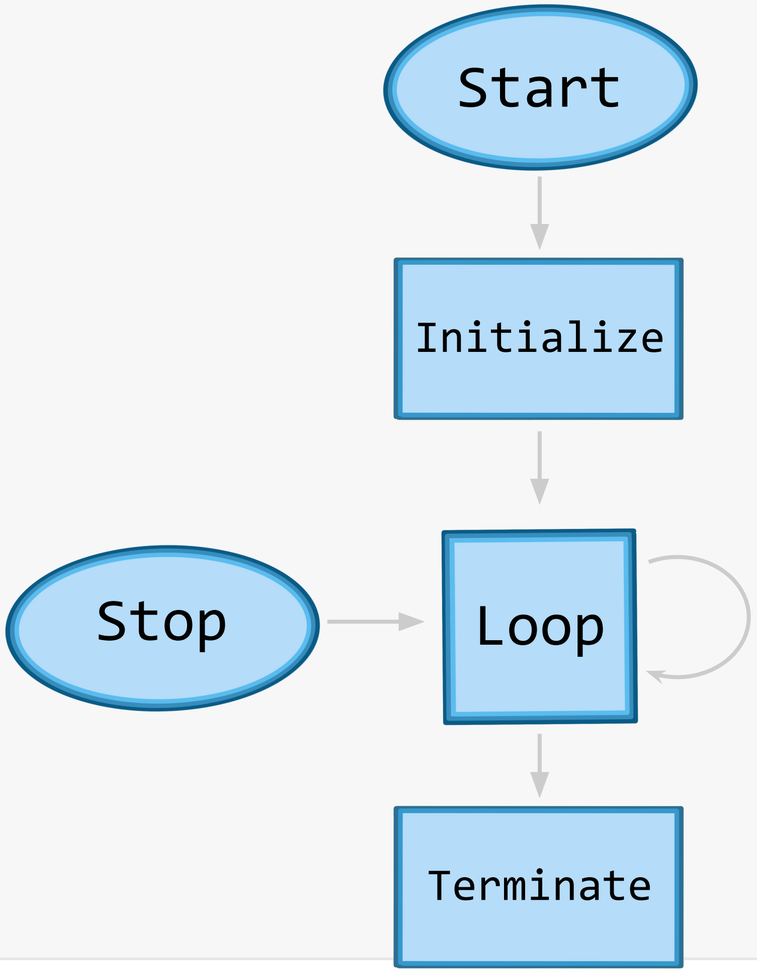
Termination

```
Mod:init/1        -> {stop, Reason}

Mod:handle_call/3 -> {stop, Reason, Reply, NewLoopData}
Mod:handle_cast/2 -> {stop, Reason, NewLoopData}

Mod:terminate(Reason, LoopData) -> Term
```

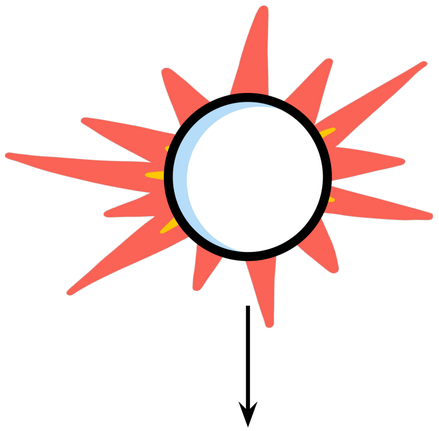
- Returning **stop** instead of **reply** / **noreply** stops the server
- The call back function **terminate/2** is called
  - It allows the server to clean up before terminating
  - Its return value is ignored

Termination

```
stop() ->
    gen_server:cast(frequency, stop).


handle_cast(stop, Frequencies) ->
    {stop, normal, Frequencies}.


terminate(Reason, {Free, Allocated}) ->
    % Clean up on termination
    ok.
```



Termination



`Mod:terminate(Reason, State)` ⟶ `Term`

- If the parent process crashes, **terminate/2** is called
  - The server must be trapping exits: **process_flag(trap_exit, true)**.
  - **Reason** will be the reason of termination

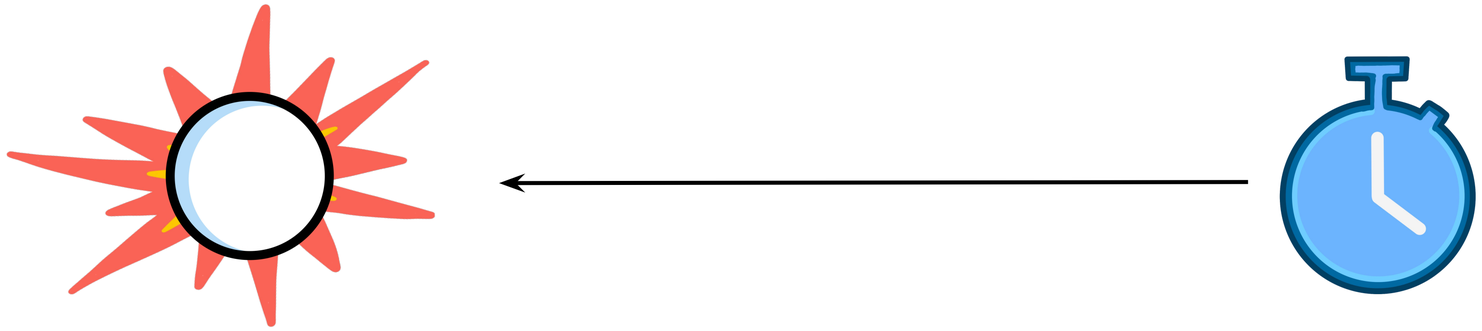
Other Messages

```
monitor_node(Node, True)
Pid ! Message
process_flag(trap_exit, true), link(Pid)
↓
Mod:handle_info(Message, LoopData)
↓
{noreply, NewLoopData}
{stop, Reason, NewLoopData}
```

- Generic servers follow a protocol hidden in the **gen_server:cast/2** and **gen_server:call/2** function calls
- Messages not following this protocol must be handled
- They are handled in the **handle_info/2** callback function

Timeouts

```
gen_server:call(Name, Message, Timeout)
```



- Timeouts can be set on client calls
- Timeout is an integer in milliseconds or the atom **infinity**
- Default timeout is 5000ms
- If the client does not receive a response, it terminates

Timeouts

```
Mod:init/1                  {ok, LoopData, Timeout}
Mod:handle_call/3   ──►     {reply, Reply, NewLoopData, Timeout}
Mod:handle_cast/2           {noreply, NewLoopData, Timeout}
Mod:handle_info/2           {noreply, NewLoopData, Timeout}

                                    ↓

                            Mod:handle_info(timeout, LoopData)
```

- Timeouts can be generated within the server
- **Timeout** is an integer in milliseconds or the atom **infinity**
- Default value is **infinity**
- Timeouts are triggered if no message is received within **Timeout** ms
- Timeouts are reset after a **timeout** event or an incoming message

Other Issues: sys module

```
sys:log(Server, Flag)        sys:log_to_file(Server,Flag)

Flag = true | {true,int()}     Flag = FileName | false
     | get | false | print
```

- **sys:log/2** turns on the logging of systems events
  - A maximum of **int()** events are logged
  - Default value is 10
- **sys:log_to_file/2** turns on the logging of system events to file

Other Issues: sys module

```
sys:statistics(Server, Flag)      sys:trace(Server,Flag)

 Flag = true | false | get         Flag = true | false
```

- **sys:statistics/2** enables the logging of statistics
  - Includes start/current times
  - Messages sent/received
  - The number of reductions
- **sys:trace/2** prints all system events to standard IO
  - You can implement your own debug/trigger function

Other Issues: statistics example

```
1> frequency:start_link().
{ok,<0.80.0>}
2> sys:statistics(frequency, true).
ok
3> frequency:allocate().
{ok,10}
4> frequency:allocate().
{ok,11}
5> frequency:deallocate(10).
ok
6> frequency:allocate().
{ok,10}
```

Other Issues: statistics example

```
7> sys:statistics(frequency, get).
{ok,[{start_time,{{Year,M,D},{H,Min,S}}},
     {current_time,{{Year,M,D},{H,Min,S2}}},
     {reductions,89},
     {messages_in,4},
     {messages_out,0}]}
```

Other Issues: sys module

```
gen_server:start_link(
   {global, Name},
   Module,
   Args,
   Options
).


Options:
{debug, [trace|statistics|log
      | {log_to_file, FileName}
      | {install, {F, State}]}
| {timeout, Time}
| {spawn_opts, Opts}
```

- Options is a list of debug functions from the sys module
- Can be set when starting the generic servers

Other Issues

```
gen_server:start_link({global, Name}, Module, Args, Opts)

gen_server:start({local, Name}, Module, Args, Options)

gen_server:start(Modules, Args, Options) or
gen_server:start_link(Module, Args, Options)
```

- **global** registers the name globally in a network of nodes
  - **gen_server:call({global, Name}, Msg)** is used to make calls
  - The name must be unique among connected nodes
  - Location transparency is preserved
- Processes do not have to be linked to their supervisors
- Processes do not have to be named
  - referred to through their Pid

Generic Servers

- Generic Servers
- Starting a Server
- Message Passing
- Termination
- Other Messages
- Timeouts
- Other Issues