Thinking in Processes

The BEAM for Developers

Erik Stenman

GNOME VILLAGE

- ✓ Gnome = Process
- ✓ Backpack = Memory
- ✓ Self-cleaning = GC
- ✓ Mail = Messages
- ✓ Scroll = Code
- ✓ Hire = Spawn
- ✓ Workbench = Scheduler
- ✓ Autonomy = Isolation
- ✓ Manager = Supervisor

Module Overview

- Schedule
- Introduction
- The Core Concepts
- Hands-On Practice
- Q&A and Discussion

Schedule

- 08:30-9:00 Registration and welcome
- 09:00 Classes start
- 10:30-11:00 Tea and Coffee Break
- 13:00-14:00 Lunch
- 15:00-15:30 Tea and Coffee Break
- 17:00 Classes finish

Coffee breaks will be served in room Lyon, on the ground floor, on the opposite side of the elevator.

Lunch will be served in the Grill restaurant in the Atrium.

Introduction

Module 1: Thinking in Processes

Duration: 60 minutes (20 min lecture, 20 min exercises, 20 min Q&A)

Learning Objectives:

- Develop a process-oriented mindset
- Understand how isolation, lightweight concurrency, and message passing create fault-tolerant systems

The Core Concepts

The Problem We're Solving

Why do distributed systems fail?

Speaker notes

Why start a BEAM internals course with philosophy?

You can learn how garbage collection works, how messages flow, how schedulers behave, but it won't make sense unless you understand *why* the BEAM works that way.

This VM was built for processes. Not threads. Not objects. Not shared-memory acrobatics. Processes. Hundreds or thousands of them. Isolated, restartable, observable.

Heap management, per-process GC, run queues, and debugging all rely on this model. If you treat it like a thread-based system, you'll fight the runtime instead of using it.

This first module is your mental reset. It's here so the rest of the course lands properly. Once you internalize this mindset, the internals won't feel abstract anymore.

The Gnome Village Metaphor

A BEAM process is like a gnome with a to-do list.

Speaker notes

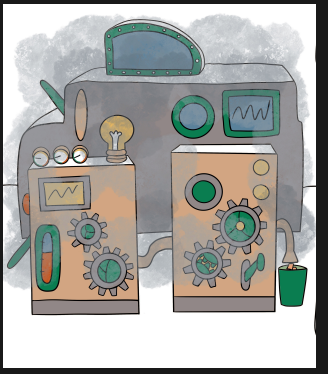
In this course, we treat every BEAM process as a "gnome" ("tomte" or "Heinzelmännchen"), a general-purpose task executor:

- A gnome reads from a to-do list (code)
- Gnomes are independent, isolated, and cheap to create
- Many gnomes can share the same task list (code reuse)
- A gnome can even switch lists (execute multiple modules)

Unlike object-oriented design, BEAM gnomes don't need to be born with a fixed role. They adapt, run, fail, restart.

We'll learn how this mental model enables massive concurrency, natural fault isolation, and surprisingly elegant code.

The Machine Park (Traditional OOP)



Objects = Shared assembly stations with threads as workers

Speaker notes

Picture a sandwich shop with shared stations. The bread station, the toppings station, the grill. Each is an object with methods like `add_ingredient()`, `set_temperature()`, `start_cooking()`.

Multiple worker threads serve different customers, all using the same stations:

```
grill.setTemperature(HIGH); grill.addItem(sandwich); grill.startCooking();
```

Looks reasonable. But here's the problem: Thread A sets the grill to HIGH for a panini. Thread B, serving a different customer, sets it to LOW for a delicate fish sandwich. Thread A's `startCooking()` runs at the wrong temperature. The panini is ruined, or worse, the fish is burnt.

Now multiply this across every station. The toppings station has one thread adding tomatoes while another is trying to close and wrap. State leaks between operations. One customer's mayo ends up on another's vegan sandwich.

You add locks around every station. Now threads wait in line. One slow order blocks everyone. A deadlock happens when Thread A holds the grill and waits for toppings, while Thread B holds toppings and waits for the grill. The whole shop freezes.

This is the shared-state trap: you have powerful machines, but threads fight over them, corrupt each other's work, and bring the system down together.

The Gnome Village (BEAM Processes)

Gnome = Process

Independent workers with private memory

Speaker notes

In the BEAM, you don't share stations. Each gnome is a process with its own workspace and tools. A gnome gets a complete order slip (message) with all the details and makes the entire sandwich from start to finish.

```
send(sandwich_gnome, {:make_sandwich, :panini, toppings: [:mozzarella, :tomato], grill: :high})
```

The gnome reads the complete order, gathers ingredients from its private stock, grills at the right temperature, and finishes the sandwich. No shared grill means no temperature conflicts. If this gnome drops the sandwich or burns it, only this one order fails. The gnome gets replaced, and the next order proceeds cleanly.

When you have 1000 customers, you spawn 1000 gnomes. Each works independently. One slow gnome doesn't block the others. One crashed gnome doesn't corrupt anyone else's sandwich. The supervisor notices, cleans up, and spawns a fresh gnome.

No locks. No shared state. No cascading failures. Just isolated workers, clear messages, and predictable outcomes. You can run millions of gnomes because they're lightweight and never interfere.

Isolation: Private Backpacks

Each gnome carries their own backpack. Memory is private.

Speaker notes

The backpack holds state that no other gnome can touch. Each process has its own heap. No shared memory. No way for another process to read or write that data.

Workbenches are shared (one per scheduler), but gnomes take turns. When a gnome gets a time slice, it steps up to the bench with its backpack, does some work, then steps aside for the next gnome. The backpack goes with the gnome, not the bench.

In the machine park, state is routed by shared wires and global settings. Multiple threads access the same objects. You need locks, which introduce contention, deadlocks, and unpredictability.

In the gnome village, isolation is the default. If you want to share data, you send a message, which copies it. The receiver gets its own copy in its own heap.

This share-nothing architecture is why BEAM runs millions of processes efficiently. No contention. No locks. No waiting for access.

The trade-off is message copying cost, but copying 100 bytes takes nanoseconds. Waiting for a lock can take milliseconds.

Private backpacks scale. Shared counters bottleneck.

Self-Cleaning Backpacks

Each gnome cleans its own backpack without pausing others.

Per-process garbage collection.

Speaker notes

When a gnome's backpack (heap) fills up, the BEAM runs GC on just that backpack. Small, fast, local pause. Other gnomes continue working at their benches.

In systems with shared heaps, GC can mean stop-the-world pauses. Everything freezes while the collector scans and compacts. Milliseconds or worse.

In the gnome village, GC pauses are isolated and short. Each backpack is small. Cleaning it is quick. The rest of the village keeps working.

This is another benefit of isolation. Not just safety, but performance. GC latency doesn't ripple across the system.

Small local pauses over global freezes.

Paper Mail, Not Direct Levers and wires

Send a letter, continue your work.

Speaker notes

Letters queue until the gnome reads them. The sender doesn't wait. The receiver processes messages when ready.

In the machine park, direct calls grab levers and block the caller. You call grill.cook() and your thread stops, waiting for completion. Synchronous. Blocking. Tight coupling.

In the gnome village, send/2 returns immediately. The message is copied to the recipient's mailbox. The sender continues. The receiver handles it later, in order, at its own pace.

Messages are immutable. You send a list of toppings, the gnome gets a copy. You can't reach into their mailbox and change it.

This decoupling makes systems resilient. A slow gnome doesn't block customers. A crashed gnome doesn't take the sender down. Pattern matching in receive lets gnomes selectively process messages.

Mailboxes buffer work. Direct calls create dependencies.

Shared Scrolls

Code is a scroll all gnomes can read. One copy, many readers, safe upgrades.

Speaker notes

One copy of code, many readers. A module is loaded once. Processes reference it but don't copy it. This is why spawning is cheap: you only allocate memory for state, not behavior.

Hot code loading works because you can swap the scroll version while workers finish current reads. Old version stays loaded until no process references it. New processes use the new version.

Update the recipe, all future sandwiches use the new instructions. Running sandwiches finish with the old recipe. Smooth, safe transition.

In the machine park, behavior is wired into parts. Rewiring risks downtime. Deployment often means restart.

In the gnome village, code upgrades can happen live. Zero-downtime releases are routine, not heroic.

Shared scrolls make processes light and upgrades safe.

Cheap to Hire

Spawning costs little in time and memory.

Lightweight workers.

Speaker notes

Kilobytes per worker, microsecond spawns. You can create tens of thousands of processes without thinking twice. A million processes on one machine is routine.

In the machine park, threads and heavy contexts limit headcount. OS threads cost megabytes. Context switching is expensive. You pool threads and reuse them, which means state leaks between tasks.

In the gnome village, processes are disposable. Spawn them freely. Let them do one task and exit. The cost is negligible.

This changes how you design. Instead of pooling workers and reusing them, you spawn a process per task, per connection, per user. One-to-one mapping. Clear ownership.

Cheap processes enable process-per-entity architecture. That architecture enables isolation, supervision, and linear scaling.

Fair Turns at the Workbench

Gnomes stand in line. Each gets a short turn.

Scheduling keeps the village responsive.

Speaker notes

One workbench per scheduler. Gnomes queue up. Each gets a short turn (measured in reductions, not wall time). After a few thousand function calls, the gnome steps aside and the next one takes the bench.

BEAM uses reduction-based scheduling. Each process gets a small number of function calls (reductions) before yielding. The scheduler switches to the next process. This happens in microseconds.

Result: regular, fair progress for many workers. No process hogs the CPU. Even if one gnome has a long task, it gets preempted and others get their turns.

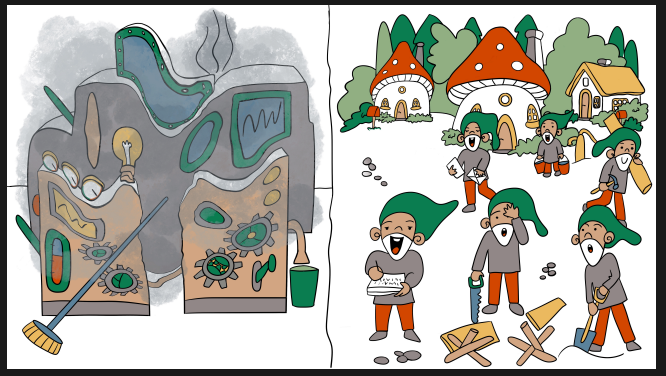
In the machine park, long critical sections starve others. One thread holding a lock blocks everyone waiting for that resource. Latency spikes. Throughput drops.

In the gnome village, cooperative scheduling spreads work evenly. Thousands of processes make incremental progress. Responsiveness stays predictable.

This is why BEAM handles soft real-time workloads well. No process monopolizes the runtime.

Failure Is Contained

One gnome's mistake affects only its output. Crashes do not corrupt neighbors.



Speaker notes

Loss is the missing result from that worker. Other processes keep running. The supervisor notices, restarts the failed worker, and life continues.

In the machine park, shared cogs carry the error into the whole machine. An exception in one object can bubble up through call stacks. Shared state gets corrupted. Threads using that state are affected.

You end up with defensive code everywhere: validate inputs, wrap in try/catch, check state constantly. Complex and still incomplete.

BEAM takes the opposite approach: let it crash. Bad data? Let the process die. The supervisor restarts it clean.

WhatsApp handles billions of messages this way. One corrupted message crashes that user's process. It restarts. Everyone else continues normally.

Isolation contains damage. Sharing spreads it.

Supervisors Are Managers

Structure work as teams with clear reporting.

Speaker notes

Managers start, monitor, and replace workers. When a worker fails, the manager notices and decides whether to restart it, restart the whole team, or escalate to a higher manager.

In the machine park, there are no managers. Only mechanical linkages. When one cog breaks, the error ripples through gears and jams the whole mechanism.

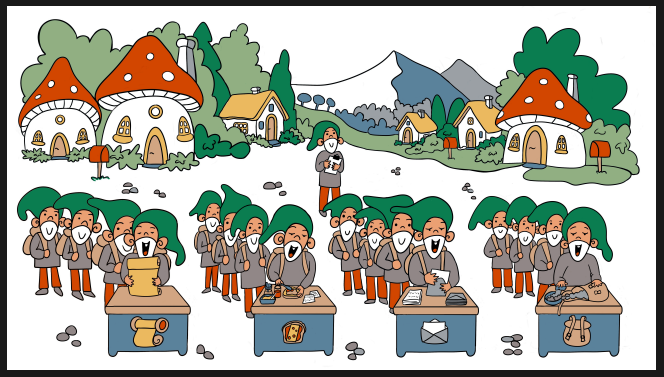
In the gnome village, supervisors define team structure. One manager can oversee panini workers, another handles salad workers. If a panini gnome crashes, only panini service restarts. Salads continue.

This hierarchy is how you build fault-tolerant systems. Errors are contained and handled at the right level. Small failures get local fixes. Widespread failures trigger coordinated restarts.

Supervision trees are not defensive programming. They are recovery design.

Summary

Build a village of gnomes, not a park of machines.



Speaker notes

The gnome village metaphor captures eight foundational principles:

- **Isolation**: Private backpacks (per-process heaps)
- **Self-cleaning**: Each gnome cleans its own backpack (per-process GC)
- **Async messaging**: Paper mail in mailboxes (message passing)
- **Shared code**: Scrolls on the shelf (hot code loading)
- **Lightweight**: Cheap to hire (spawn costs microseconds)
- **Fair scheduling**: Turns at the workbench (reduction-based preemption)
- **Fault isolation**: One crash stays local (contained failures)
- **Supervision**: Managers replace workers (recovery trees)

The metaphor maps directly to runtime behavior. Workbenches are schedulers. Backpacks are heaps. Scrolls are modules. Mailboxes are message queues.

This mental model makes the rest of the course concrete:

- Module 2: Data costs (what goes in messages and backpacks)
- Module 3: Memory management (when backpacks get cleaned)
- Module 4: Scheduler behavior (how workbench queues operate)
- Module 5: Debugging tools (watching gnomes work)
- Module 6: Patterns (lifecycle, backpressure, message design, observability)
- Module 7: Projects (apply the village design to real systems)

Hands-On Practice

Interactive Exercises

Open the LiveBook:

`module-1-exercises.livemd`

20 minutes of hands-on practice with:

- Process spawning and messaging
- Fault isolation
- Mailbox observation
- System design

Speaker notes

The exercises are in an interactive LiveBook notebook. Students should open `module-1-exercises.livemd` in LiveBook to work through the hands-on exercises below.

Each exercise has runnable code cells and discussion questions.

Question 1

Why does per-process garbage collection prevent global pauses?

- Each process has its own heap and GC runs independently
- GC is disabled in the BEAM
- A GC pause in one process doesn't block other processes
- Processes share a heap but GC runs in parallel

Question 2

In OOP with shared state, what problem does the BEAM avoid through process isolation?

- Race conditions on shared mutable data
- Deadlocks from lock contention
- Memory leaks from garbage collection
- Slow message passing

Question 3

How does the BEAM scheduler ensure fairness among processes?

- Each process gets exactly 10ms of CPU time
- Processes are preempted after consuming a reduction budget
- Processes voluntarily yield when they call receive
- Reduction counting allows fair scheduling without timer interrupts

Question 4

What is the trade-off of process isolation (message copying)?

- Safety and fault tolerance at the cost of copying overhead
- Faster performance but less reliability
- Lower memory usage but higher CPU usage
- No shared state bugs, but measurable copy cost for large data

Question 5

When would you use ETS instead of message passing between processes?

- For shared read-mostly configuration data
- For all inter-process communication
- When copy cost outweighs isolation benefits
- Never, ETS breaks the actor model

Question 6

In a process-oriented architecture, where should state live?

- In individual process heaps when each entity owns its state
- In ETS or databases when state must survive process restarts
- In a global shared data structure protected by locks
- Distributed across processes with clear ownership boundaries

Question 7

What happens when you force GC on a process with a 100MB heap?

- All processes pause until GC completes
- Only that process pauses; others continue working
- The VM triggers a global GC
- The process crashes if GC takes too long

Question 8

Why use reduction-based scheduling instead of time-based preemption?

- Ensures processes yield at safe, known points in execution
- Avoids complexity of interrupting arbitrary VM state
- Reduction counting is more accurate for fairness
- Reduction counting uses less CPU overhead

Question 9

In the OOP order processing example, what does the BEAM approach replace locks with?

- Faster locks using atomic operations
- Message passing through isolated process mailboxes
- Optimistic concurrency with retry logic
- Serialization of updates through a single process

Question 10

What is the key architectural difference between the "machine park" and "gnome village"?

- Machines share state and require locks; gnomes have private state
- Machine failures cascade; gnome failures are isolated
- Machines are faster but less reliable
- Gnomes use more memory per worker

Q&A and Discussion

1. "Why not just use threads?"

- What makes BEAM processes different?
- When would threads be better?

2. "What about shared state?"

- How do you handle state that multiple processes need?
- When is message passing too expensive?

3. "How do supervisors know what to restart?"

- What restart strategies exist?
- How deep should supervision trees go?

Navigation

Back to Course Structure | Next Module: Data Types & Message Costs →