

Learn You Some Erlang

for great good!

by Frederic Trottier-Hebert

Except where otherwise noted, content on this book is licensed under a Creative Commons Attribution Non-Commercial No Derivative License.



Table of Contents

Introduction

- [About this tutorial](#)
- [So what's Erlang?](#)
- [Don't drink too much Kool-Aid](#)
- [What you need to dive in](#)
- [Where to get help](#)

Starting Out

- [The Shell](#)
- [Shell Commands](#)

Starting Out (for real)

- [Numbers](#)
- [Invariable Variables](#)
- [Atoms](#)
- [Boolean Algebra and Comparison Operators](#)
- [Tuples](#)
- [Lists](#)
- [List Comprehensions](#)
- [Bit Syntax](#)
- [Binary Comprehensions](#)

Modules

- [What are modules](#)

- [Module Declaration](#)
- [Compiling the Code](#)
- [More About Modules](#)

Syntax in Functions

- [Pattern Matching](#)
- [Guards, Guards!](#)
- [What the If!?](#)
- [In Case ... of](#)
- [Which to use?](#)

Types (or lack thereof)

- [Dynamite-strong Typing](#)
- [Type Conversions](#)
- [To Guard a Data Type](#)
- [For Type Junkies](#)

Recursion

- [Hello recursion!](#)
- [Length](#)
- [Length of Tail Recursion](#)
- [More recursive functions](#)
- [Quick, Sort!](#)
- [More than lists](#)
- [Thinking recursively](#)

Higher Order Functions

- [Let's get functional](#)
- [Anonymous functions](#)
- [Maps, filters, folds and more](#)

Errors and Exceptions

- [Not so fast!](#)
- [A Compilation of Errors](#)
- [No, YOUR logic is wrong!](#)
- [Run-time Errors](#)
- [Raising Exceptions](#)
- [Dealing with Exceptions](#)
- [Wait, there's more!](#)
- [Try a try in a tree](#)

Functionally Solving Problems

- [Reverse Polish Notation Calculator](#)
- [Heathrow to London](#)

A Short Visit to Common Data Structures

- [Won't be too long, promised!](#)
- [Records](#)
- [Key-Value Stores](#)
- [Arrays](#)
- [A Set of Sets](#)
- [Directed Graphs](#)
- [Queues](#)
- [End of the short visit](#)

The Hitchhiker's Guide to Concurrency

- [Don't Panic](#)
- [Concepts of Concurrency](#)
- [Not Entirely Unlike Linear Scaling](#)
- [So long and thanks for all the fish!](#)

More On Multiprocessing

- [State Your State](#)
- [We love messages, but we keep them secret](#)
- [Time Out](#)
- [Selective Receives](#)

Errors and Processes

- [Links](#)
- [It's a Trap!](#)
- [Monitors](#)
- [Naming Processes](#)

Designing a Concurrent Application

- [Understanding the Problem](#)
- [Defining the Protocol](#)
- [Lay Them Foundations](#)
- [An Event Module](#)
- [The Event Server](#)
- [Hot Code Loving](#)
- [I Said, Hide Your Messages](#)
- [A Test Drive](#)

- [Adding Supervision](#)
- [Namespaces \(or lack thereof\)](#).

What is OTP?

- [It's The Open Telecom Platform!](#)
- [The Common Process, Abstracted](#)
- [The Basic Server](#)
- [Specific Vs. Generic](#)

Clients and Servers

- [Callback to the Future](#)
- [.BEAM me up, Scotty!](#)

Rage Against The Finite-State Machines

- [What Are They?](#)
- [Generic Finite-State Machines](#)
- [A Trading System Specification](#)
- [Game trading between two players](#)
- [That Was Quite Something](#)
- [Fit for the Real World?](#)

Event Handlers

- [Handle This! *pumps shotgun*](#)
- [Generic Event Handlers](#)
- [It's Curling Time!](#)
- [Alert The Press!](#)

Who Supervises The Supervisors?

- [From Bad to Good](#)
- [Supervisor Concepts](#)
- [Using Supervisors](#)
- [Child Specifications](#)
- [Testing it Out](#)
- [Dynamic Supervision](#)

Building an Application With OTP

- [A Pool of Processes](#)
- [The Onion Layer Theory](#)
- [A Pool's Tree](#)
- [Implementing the Supervisors](#)
- [Working on the Workers](#)
- [Writing a Worker](#)
- [Run Pool Run](#)
- [Cleaning the Pool](#)

Building OTP Applications

- [Why Would I Want That?](#)
- [My Other Car is a Pool](#)
- [The Application Resource File](#)
- [The Application Behaviour](#)
- [From Chaos to Application](#)
- [Library Applications](#)

The Count of Applications

- [From OTP Application to Real Application](#)
- [Run App Run](#)

- [Included Applications](#)
- [Complex Terminations](#)

Release is the Word

- [Am I an Executable Yet?](#)
- [Fixing The Leaky Pipes](#)
- [Releases With Systools](#)
- [Releases With Reltool](#)
- [Recipes](#)
- [Released From Releases](#)

Leveling Up in The Process Quest

- [The Hiccups of Appups and Relups](#)
- [The 9th Circle of Erl](#)
- [Progress Quest](#)
- [Making Process Quest Better](#)
- [Appup Files](#)
- [Upgrading the Release](#)

Buckets Of Sockets

- [IO Lists](#)
- [TCP and UDP: Bro-tocols](#)
- [UDP Sockets](#)
- [TCP Sockets](#)
- [More Control With Inet](#)
- [Sockserv, Revisited](#)
- [Where to go From Now?](#)

EUnited Nations Council

- [The Need for Tests](#)
- [EUnit, What's a EUnit?](#)
- [Test Generators](#)
- [Fixtures](#)
- [Testing Regis](#)
- [He Who Knits Eunits](#)

Bears, ETS, Beets

- [The Concepts of ETS](#)
- [ETS Phone Home](#)
- [Meeting Your Match](#)
- [You Have Been Selected](#)
- [DETS](#)
- [A Little Less Conversation, A Little More Action Please](#)

Distribunomicon

- [Alone in the Dark](#)
- [This is my Boomstick](#)
- [Fallacies of Distributed Computing](#)
- [Dead or Dead Alive](#)
- [My Other CAP is a Theorem](#)
- [Setting up an Erlang Cluster](#)
- [Cookies](#)
- [Remote Shells](#)
- [Hidden Nodes](#)
- [The Walls are Made of Fire and the Goggles do Nothing](#)

- [The Calls from Beyond](#)
- [Burying the Distribunomicon](#)

Distributed OTP Applications

- [Adding More to OTP](#)
- [Taking and Failing Over](#)
- [The Magic 8-Ball](#)
- [Making the Application Distributed](#)

Common Test for Uncommon Tests

- [What is Common Test](#)
- [Common Test Cases](#)
- [Testing With State](#)
- [Test Groups](#)
- [The Meeting Room](#)
- [Test Suites](#)
- [Test Specifications](#)
- [Large Scale Testing](#)
- [Integrating EUnit within Common Test](#)
- [Is There More?](#)

Mnesia And The Art of Remembering

- [What's Mnesia](#)
- [What Should the Store Store](#)
- [From Record to Table](#)
- [Of Schemas and Mnesia](#)
- [Creating Tables for Real](#)
- [Access and Context](#)

- [Reads, Writes, and More](#)
- [Implementing The First Requests](#)
- [Accounts And New Needs](#)
- [Meet The Boss](#)
- [Deleting Stuff, Demonstrated](#)
- [Query List Comprehensions](#)
- [Remember Mnesia](#)

Type Specifications and Erlang

- [PLT Are The Best Sandwiches](#)
- [Success Typing](#)
- [Type Inference and Discrepancies](#)
- [Typing About Types of Types](#)
- [Typing Functions](#)
- [Typing Practice](#)
- [Exporting Types](#)
- [Typed Behaviours](#)
- [Polymorphic Types](#)
- [You're my Type](#)

Conclusion

- [A Few Words](#)
- [Other Topics](#)
- [LYSE as a book](#)

Postscript: Maps

- [About This Chapter](#)
- [EEP, EEP!](#)

- [What Maps Shall Be](#)
- [Stubby Legs for Early Releases](#)
- [Mexican Standoff](#)
- [How This Book Would Be Revised For Maps](#)

Postscript: Time Goes On

- [On Time for Time](#)
- [How Things Were](#)
- [How Things Are \(18.0+\)](#)
- [Time Warp](#)
- [How to Survive Time Warps](#)

Introduction

About this tutorial

This is the beginning of Learn You Some Erlang for Great Good! Reading this tutorial should be one of your first steps in learning Erlang, so let's talk about it a bit.



First of all, I began growing the idea of writing this after reading Miran Lipovača's Learn You a Haskell for great Good! (LYAH) tutorial; I thought he did a great job making the language attractive and the learning experience friendly. As I already knew him, I asked him how he felt about me writing an Erlang version of his book. He liked the idea, being somewhat interested in Erlang.

So here I am typing this. Of course there were other sources to my motivation: I mainly find the entry to the language to be hard (the web has sparse documentation and otherwise you need to buy books), and I thought the community would benefit from a LYAH-like guide. Less importantly, I've seen people attributing Erlang too much or not enough merit sometimes based on sweeping generalizations. Then there are people who sure as hell believe Erlang is nothing but

hype. If I'd like to convince them otherwise, I know they're not likely to read this in the first place.

This book thus wants itself to be a way to learn Erlang for people who have basic knowledge of programming in imperative languages (such as C/C++, Java, Python, Ruby, etc) and may or may not know functional programming (Haskell, Scala, Erlang, Clojure, OCaml...). I also want to write this book in a honest manner, selling Erlang for what it is, acknowledging its weaknesses and strengths.

So what's Erlang?

First of all, Erlang is a functional programming language. If you have ever worked with imperative languages, statements such as `i++` may be normal to you; in functional programming they are not allowed. In fact, changing the value of any variable is strictly forbidden! This may sound weird at first, but if you remember your math classes, it's in fact how you've learned it:

```
y = 2  
x = y + 3  
x = 2 + 3  
x = 5
```

Had I added the following:

```
x = 5 + 1  
x = x  
5 = 6
```

You would have been very confused. Functional programming recognizes this: If I say x is 5, then I can't logically claim it is also 6! This would be dishonest. This is also why a function with the same parameter should always return the same result:

```
x = add_two_to(3) = 5  
x = 5
```

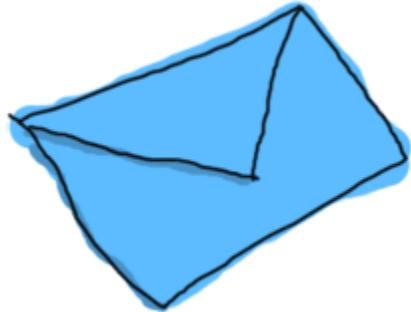
Functions always returning the same result for the same parameter is called referential transparency. It's what lets us replace `add_two_to(3)` with 5, as the result of $3+2$ will always be 5. That means we can then glue dozens of functions together in order to resolve more complex problems while being sure nothing will break. Logical and clean isn't it? There's a problem though:

```
x = today() = 2009/10/22  
-- wait a day --  
x = today() = 2009/10/23  
x = x  
2009/10/22 = 2009/10/23
```

Oh no! My beautiful equations! They suddenly all turned wrong! How come my function returns a different result every day?

Obviously, there are some cases where it's useful to break referential transparency. Erlang has this very pragmatic approach with functional programming: obey its purest principles (referential transparency, avoiding mutable data,

etc), but break away from them when real world problems pop up.



Now, we defined Erlang as a functional programming language, but there's also a large emphasis on concurrency and high reliability. To be able to have dozens of tasks being performed at the same time, Erlang uses the actor model, and each actor is a separate process in the virtual machine. In a nutshell, if you were an actor in Erlang's world, you would be a lonely person, sitting in a dark room with no window, waiting by your mailbox to get a message. Once you get a message, you react to it in a specific way: you pay the bills when receiving them, you respond to Birthday cards with a "Thank you" letter and you ignore the letters you can't understand.

Erlang's actor model can be imagined as a world where everyone is sitting alone in their own room and can perform a few distinct tasks. Everyone communicates strictly by writing letters and that's it. While it sounds like a boring life (and a new age for the postal service), it means you can ask many people to perform very specific tasks for you, and none of them will ever do something wrong or make

mistakes which will have repercussions on the work of others; they may not even know the existence of people other than you (and that's great).

To escape this analogy, Erlang forces you to write actors (processes) that will share no information with other bits of code unless they pass messages to each other. Every communication is explicit, traceable and safe.

When we defined Erlang, we did so at a language level, but in a broader sense, this is not all there is to it: Erlang is also a development environment as a whole. The code is compiled to bytecode and runs inside a virtual machine. So Erlang, much like Java and kids with ADD, can run anywhere. The standard distribution includes (among others) development tools (compiler, debugger, profiler, test framework), the Open Telecom Platform (OTP) Framework, a web server, a parser generator, and the mnesia database, a key-value storage system able to replicate itself on many servers, supporting nested transactions and letting you store any kind of Erlang data.

The VM and the libraries also allow you to update the code of a running system without interrupting any program, distribute your code with ease on many computers and manage errors and faults in a simple but powerful manner.



We'll see how to use most of these tools and achieve safety later on, but for now, I'll tell you about a related general policy in Erlang: Let it crash. Not like a plane with dozens of passengers dying, but more like a tightrope walker with a safety net under him. While you should avoid making mistakes, you won't need to check for every type or error condition in most cases.

Erlang's ability to recover from errors, organize code with actors and making it scale with distribution and concurrency all sound awesome, which brings us to the next section...

Don't drink too much Kool-Aid

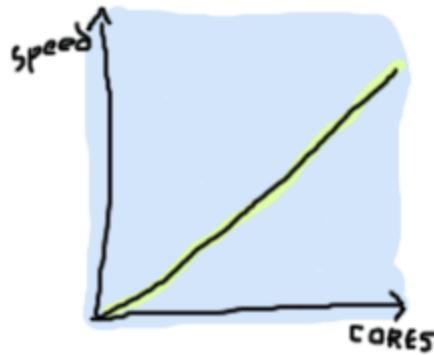
There may be many little yellowish-orange sections named like this one around the book (you'll recognize them when you see them). Erlang is currently gaining lots of popularity due to zealous talks which may lead people to believe it's

more than what it really is. These reminders will be there to help you keep your feet on the ground if you're one of these overenthusiastic learners.

The first case of this is related to Erlang's massive scaling abilities due to its lightweight processes. It is true that Erlang processes are very light: you can have hundreds of thousands of them existing at the same time, but this doesn't mean you have to use it that way just because you can. For example, creating a shooter game where everything including bullets is its own actor is madness. The only thing you'll shoot with a game like this is your own foot. There is still a small cost in sending a message from actor to actor, and if you divide tasks too much, *you will make things slower!*

I'll cover this with more depth when we're far enough into the learning to actually worry about it, but just keep in mind that randomly throwing parallelism at a problem is not enough to make it go fast. Don't be sad; there are times when using hundreds of processes will both be possible and useful! It's just not happening all the time.

Erlang is also said to be able to scale in a directly proportional manner to how many cores your computer has, but this is usually not true: it is possible, but most problems do not behave in a way that lets you just run everything at the same time.



There's something else to keep in mind: while Erlang does some things very well, it's technically still possible to get the same results from other languages. The opposite is also true; evaluate each problem as it needs to be, and choose the right tool according to the problem being addressed. Erlang is no silver bullet and will be particularly bad at things like image and signal processing, operating system device drivers, etc. and will shine at things like large software for server use (i.e.: queues, map-reduce), doing some lifting coupled with other languages, higher-level protocol implementation, etc. Areas in the middle will depend on you. You should not necessarily wall yourself in server software with Erlang: there have been cases of people doing unexpected and surprising things. One example is IANO, a robot created by the UNICT team, which uses Erlang for its artificial intelligence and won the silver medal at the 2009 eurobot competition. Another example is Wings 3D, an open source 3D modeler (but not a renderer) written in Erlang and thus cross-platform.

What you need to dive in

All you need to get started is a text editor and the Erlang environment. You can get the source code and the Windows binaries from the official Erlang website. I won't go into much installation details, but for Windows, just download and run the binary files. Don't forget to add your Erlang directory to your *PATH* system variable to be able to access it from the command line.

On Debian-based Linux distributions, you should be able to install the package by doing `$ apt-get install erlang`. On Fedora (if you have 'yum' installed), you can achieve the same by typing `# yum install erlang`. However, these repositories often hold outdated versions of the Erlang packages; Using an outdated version could give you some differences with what you'll get from this tutorial and a hit in performance with certain applications. I thus encourage you to compile from source. Consult the README file within the package and Google to get all the installing details you'll need, they'll do a far better job than I ever will.

On FreeBSD, many options are available to you. If you're using portmaster, you can do `portmaster lang/erlang`. For standard ports, it should be `cd /usr/ports/lang/erlang; make install clean`. Finally, if you want to use packages, run `pkg_add -rv erlang`.

If you're on OSX, you can install Erlang with `$ brew install erlang` (with Homebrew) or by doing `$ port install erlang` (if you prefer MacPorts.)

Alternatively, Erlang Solutions Ltd. offers packages for all major OSes, which generally work pretty well (pick a 'Standard' distribution).

Note: at the time of this writing, I'm using Erlang version R13B+, but for best results, you should use newer ones.

Where to get Help

There are a few places where you can get help. If you're using linux, you can access the man pages for good technical documentation. Erlang has a lists module (which we'll soon see): to get the documentation on lists, just type in \$ erl -man lists.

On Windows, the installation should include HTML documentation. You can download it at any time from the official erlang site, or consult one of the cleaner alternative sites.

Good coding practices can be found here once you feel you need to get everything clean. The code in this book will attempt to follow these guidelines, too.

Now, there are times where just getting the technical details isn't enough. When that happens, I tend to turn to two main sources: the official mailing list (you should follow it just to learn a bunch) and the #erlang channel on irc.freenode.net.

Oh and if you're the type of person to go for cookbooks and pre-made recipes, trapexit is the place you're looking for.

They also mirror the mailing lists as a forum and a general wiki, which can always be helpful.

Starting Out

The Shell

In Erlang, you can test most of your stuff in an emulator; it will run your scripts when compiled and deployed, but it will also let you edit stuff live. To start the shell in Linux, open a terminal and then type in \$ erl. If you've set up everything fine, you should see text like this:

```
Erlang R13B01 (erts-5.7.2) [source] [smp:2:2] [rq:2] [async-threads:0] [hipe] [kernel-poll:false]
```

```
Eshell V5.7.2 (abort with ^G)
```

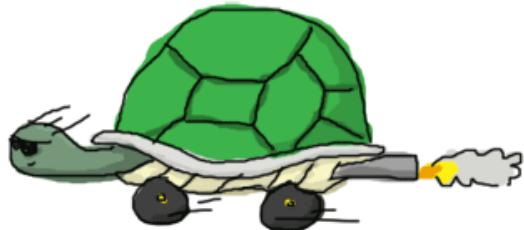
Congratulations, you're running the Erlang shell!

For Windows users, you can still run the erl.exe shell, but it's recommended you instead use werl.exe, which can be found in your start menu (programs > Erlang). Werl is a windows-only implementation of the Erlang shell, having its own window with scrollbars and supporting command-line editing (like copy-pasting, which got to be a pain with the standard cmd.exe shell in Windows). The erl shell is still required if you want to redirect standard input or output, or use pipelines.

We'll be able to enter and run code in the emulator, but first, let's see how we can get around in it.

Shell Commands

The Erlang shell has a built-in line editor based on a subset of Emacs, a popular text editor that's been in use since the 70s. If you know Emacs, you should be fine. For the others, you'll do fine anyway.



First of all, if you type some text and then go `^A` (`Ctrl+A`), you should see your cursor moving to the beginning of the line. `^E` (`Ctrl+E`) gets you to the end. You can use arrow keys to go forward, backwards, show previous or next lines so you can repeat code.

If you type something like `li` and then press "tab", the shell will have completed the terms for you to `lists`. Press tab again, and the shell will suggest you many functions to use after. This is Erlang completing the module `lists` and then suggesting functions from it. You may find the notation weird, but don't worry, you'll get familiar with it soon enough.

I think we've seen enough of shell functionality to be alright, except for one thing: we don't know how to leave! There's a fast way to find how. Just type in `help()`. and you should get information on a bunch of commands you can use in the shell (do not forget the full stop `(.)` as it is necessary for the command to run). We'll use some of them at a later point, but the only line of concern to us in order to get out is

`q()` -- quit - shorthand for `init:stop()`

So this is one way to do it (in fact, two ways). But this won't help us if the shell freezes! If you were paying attention, when you started the shell, there was a comment about 'aborting with `^G`'. Let's do that, and then press `h` to get help!

User switch command

```
--> h
c [nn]      - connect to job
i [nn]      - interrupt job
k [nn]      - kill job
j           - list all jobs
s [shell]   - start local shell
r [node [shell]] - start remote shell
```

```
q      - quit erlang  
? | h      - this message  
-->
```

If you type in i then c, Erlang should stop the currently running code and bring you back to a responsive shell. j will give you a list of processes running (a star after a number indicates this is the job you are currently running), which you can then interrupt with i followed by the number. If you use k, you will kill the shell as it is instead of just interrupting it. Press s to start a new one.

```
Eshell V5.7.2 (abort with ^G)  
1> "OH NO THIS SHELL IS UNRESPONSIVE!!! *hits ctrl+G*"  
User switch command  
--> k  
--> c  
Unknown job  
--> s  
--> j  
  2* {shell,start,[]}  
--> c 2  
Eshell V5.7.2 (abort with ^G)  
1> "YESS!"
```

If you read back the help text, you'll notice we can start remote shells. I won't get into details right now, but this should give you an idea of what the Erlang VM can do apart from running code. For now, let's get things started (for real).

Starting Out (for real)

Erlang is a relatively small and simple language (in the way C is simpler than C++). There are a few basic data types built in the language, and as such, this chapter will cover most of them. Reading it is strongly advised as it explains the building blocks for all the programs you'll write with Erlang later on.

Numbers

In the Erlang shell, **expressions have to be terminated with a period followed by whitespace** (line break, a space etc.), otherwise they won't be executed. You can separate expressions with commas, but only the result of the last one will be shown (the others are still executed). This is certainly unusual syntax for most people and it comes from the days Erlang was implemented directly in Prolog, a logic programming language.

Open the Erlang shell as described in the previous chapters and let's type them things!

```
1> 2 + 15.  
17  
2> 49 * 100.  
4900  
3> 1892 - 1472.  
420  
4> 5 / 2.  
2.5  
5> 5 div 2.  
2  
6> 5 rem 2.  
1
```

You should have noticed Erlang doesn't care if you enter floating point numbers or integers: both types are supported when dealing with

arithmetic. Integers and floating values are pretty much the only types of data Erlang's mathematical operators will handle transparently for you. However, if you want to have the integer-to-integer division, use `div`, and to have the modulo operator, use `rem` (remainder).



Note that we can use several operators in a single expression, and mathematical operations obey the normal precedence rules.

```
7> (50 * 100) - 4999.  
1  
8> -(50 * 100 - 4999).  
-1  
9> -50 * (100 - 4999).  
244950
```

If you want to express integers in other bases than base 10, just enter the number as `Base#Value` (given Base is in the range 2..36):

```
10> 2#101010.  
42  
11> 8#0677.  
447  
12> 16#AE.  
174
```

Awesome! Erlang has the power of the calculator you have on the corner of your desk with a weird syntax on top of it! Absolutely exciting!

Invariable Variables

Doing arithmetic is alright, but you won't go far without being able to store results somewhere. For that, we'll use variables. If you have read the intro to this book, you'll know that variables can't be variable in functional

programming. The basic behavior of variables can be demonstrated with these 7 expressions (note that variables begin with an uppercase letter):

```
1> One.  
* 1: variable 'One' is unbound  
2> One = 1.  
1  
3> Un = Uno = One = 1.  
1  
4> Two = One + One.  
2  
5> Two = 2.  
2  
6> Two = Two + 1.  
** exception error: no match of right hand side value 3  
7> two = 2.  
** exception error: no match of right hand side value 2
```

The first thing these commands tell us is that you can assign a value to a variable exactly once; then you can 'pretend' to assign a value to a variable if it's the same value it already has. If it's different, Erlang will complain. It's a correct observation, but the explanation is a bit more complex and depends on the = operator. The = operator (not the variables) has the role of comparing values and complaining if they're different. If they're the same, it returns the value:

```
8> 47 = 45 + 2.  
47  
9> 47 = 45 + 3.  
** exception error: no match of right hand side value 48
```

What this operator does when mixed with variables is that if the left-hand side term is a variable and it is unbound (has no value associated to it), Erlang will automatically bind the right-hand side value to the variable on the left-hand side. The comparison will consequently succeed and the variable will keep the value in memory.

This behavior of the = operator is the basis of something called 'Pattern matching', which many functional programming languages have, although Erlang's way of doing things is usually regarded as more flexible and complete than alternatives. We'll see pattern matching with more detail when we visit the tuple and list types in this very chapter, and also with functions in the following chapters.

The other thing the commands 1-7 told us is that variable names must begin with a capital letter. Command 7 failed because the word *two* had a lowercase letter to begin with. Technically, variables can start with an underscore ('_') too, but by convention their use is restricted to values you do not care about, yet you felt it was necessary to document what it contains.

You can also have variables that are only an underscore:

```
10> _ = 14+3.  
17  
11> _.  
* 1: variable '_' is unbound
```

Unlike any other kind of variable, it won't ever store any value. Totally useless for now, but you'll know it exists when we need it.

Note: If you're testing in the shell and save the wrong value to a variable, it is possible to 'erase' that variable by using the function f(variable).. If you wish to clear all variable names, do f()..

These functions are there only to help you when testing and only work in the shell. When writing real programs, we won't be able to destroy values that way. Being able to do it only in the shell makes sense if you acknowledge Erlang being usable in industrial scenarios: it is wholly possible to have a shell being active for years without interruption... Let's bet that the variable X would be used more than once in that time period.

Atoms

There is a reason why variables names can't begin with a lowercase character: atoms. Atoms are literals, constants with their own name for value. What you see is what you get and don't expect more. The atom *cat* means "cat" and that's it. You can't play with it, you can't change it, you can't smash it to pieces; it's *cat*. Deal with it.

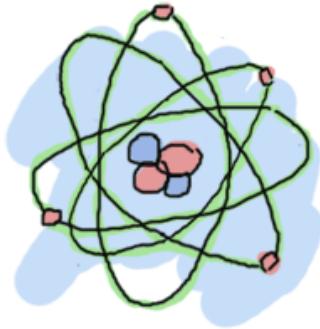
While single words starting with a lowercase letter is a way to write an atom, there's more than one manner to do it:

```
1> atom.  
atom  
2> atoms_rule.  
atoms_rule  
3> atoms_rule@erlang.  
atoms_rule@erlang  
4> 'Atoms can be cheated!'.  
'Atoms can be cheated'  
5> atom = 'atom'.  
atom
```

An atom should be enclosed in single quotes ('') if it does not begin with a lower-case letter or if it contains other characters than alphanumeric characters, underscore (_), or @.

Expression 5 also shows that an atom with single quotes is exactly the same as a similar atom without them.

I compared atoms to constants having their name as their values. You may have worked with code that used constants before: as an example, let's say I have values for eye colors: BLUE -> 1, BROWN -> 2, GREEN -> 3, OTHER -> 4. You need to match the name of the constant to some underlying value. Atoms let you forget about the underlying values: my eye colors can simply be 'blue', 'brown', 'green' and 'other'. These colors can be used anywhere in any piece of code: the underlying values will never clash and it is impossible for such a constant to be undefined! If you really want constants with values associated to them, there's a way to do it that we'll see in [chapter 4](#) (Modules).



An atom is therefore mainly useful to express or qualify data coupled with it. Used alone, it's a bit harder to find a good use to it. This is why we won't spend more time toying with them; their best use will come when coupled with other types of data.

Don't drink too much Kool-Aid:

Atoms are really nice and a great way to send messages or represent constants. However there are pitfalls to using atoms for too many things: an atom is referred to in an "atom table" which consumes memory (4 bytes/atom in a 32-bit system, 8 bytes/atom in a 64-bit system). The atom table is not garbage collected, and so atoms will accumulate until the system tips over, either from memory usage or because 1048577 atoms were declared.

This means atoms should not be generated dynamically for whatever reason; if your system has to be reliable and user input lets someone crash it at will by telling it to create atoms, you're in serious trouble.

Atoms should be seen as tools for the developer because honestly, it's what they are.

Note: some atoms are reserved words and can not be used except for what the language designers wanted them to be: function names, operators, expressions, etc. These are: after and also band begin bnot bor bsl bsr bxor case catch cond div end fun if let not of or orelse query receive rem try when xor



Boolean Algebra & Comparison operators

One would be in pretty deep trouble if one couldn't tell the difference between what's small and big, what's true and false. As any other language, Erlang has ways to let you use boolean operations and to compare items.

Boolean algebra is dirt simple:

1> true and false.

false

2> false or true.

true

3> true xor false.

true

4> not false.

true

5> not (true and true).

false

Note: the boolean operators and and or will always evaluate arguments on both sides of the operator. If you want to have the short-circuit operators (which will only evaluate the right-side argument if it needs to), use andalso and orelse.

Testing for equality or inequality is also dirt simple, but has slightly different symbols from those you see in many other languages:

6> 5 == 5.

true

```
7> 1 =:= 0.  
false  
8> 1 /= 0.  
true  
9> 5 =:= 5.0.  
false  
10> 5 == 5.0.  
true  
11> 5 /= 5.0.  
false
```

First of all, if your usual language uses `==` and `!=` to test for and against equality, Erlang uses `=:=` and `/=`. The three last expressions (lines 9 to 11) also introduce us to a pitfall: Erlang won't care about floats and integers in arithmetic, but will do so when comparing them. No worry though, because the `==` and `/=` operators are there to help you in these cases. This is important to remember whether you want exact equality or not.

Other operators for comparisons are `<` (less than), `>` (greater than), `>=` (greater than or equal to) and `=<` (less than or equal to). That last one is backwards (in my opinion) and is the source of many syntax errors in my code. Keep an eye on that `=<`.

```
12> 1 < 2.  
true  
13> 1 < 1.  
false  
14> 1 >= 1.  
true  
15> 1 =< 1.  
true
```

What happens when doing `5 + llama` or `5 == true`? There's no better way to know than trying it and subsequently getting scared by error messages!

```
12> 5 + llama.  
** exception error: bad argument in an arithmetic expression  
in operator +/2  
called as 5 + llama
```

Welp! Erlang doesn't really like you misusing some of its fundamental types! The emulator returns a nice error message here. It tells us it doesn't like one of the two arguments used around the `+` operator!

Erlang getting mad at you for wrong types is not always true though:

```
13> 5 =:= true.  
false
```

Why does it refuse different types in some operations but not others? While Erlang doesn't let you *add* anything with everything, it will let you *compare* them. This is because the creators of Erlang thought pragmaticism beats theory and decided it would be great to be able to simply write things like general sorting algorithms that could order any term. It's there to make your life simpler and can do so the vast majority of the time.

There is one last thing to keep in mind when doing boolean algebra and comparisons:

```
14> 0 == false.  
false  
15> 1 < false.  
true
```

Chances are you're pulling your hair if you come from procedural languages or most object-oriented languages. Line 14 should evaluate to *true* and line 15 to *false*! After all, *false* means 0 and *true* is anything else! Except in Erlang. Because I lied to you. Yes, I did that. Shame on me.

Erlang has no such things as boolean *true* and *false*. The terms *true* and *false* are atoms, but they are integrated well enough into the language you shouldn't have a problem with that as long as you don't expect *false* and *true* to mean anything but *false* and *true*.

Note: The correct ordering of each element in a comparison is the following:

number < atom < reference < fun < port < pid < tuple < list < bit string

You don't know all these types of things yet, but you will get to know them through the book. Just remember that this is why you can compare anything with anything! To quote Joe Armstrong, one of the creators of Erlang: "The actual order is not important - but that a total ordering is well defined is important."

Tuples

A tuple is a way to organize data. It's a way to group together many terms when you know how many there are. In Erlang, a tuple is written in the form {Element1, Element2, ..., ElementN}. As an example, you'd give me the coordinates (x,y) if you wanted to tell me the position of a point in a Cartesian graph. We can represent this point as a tuple of two terms:

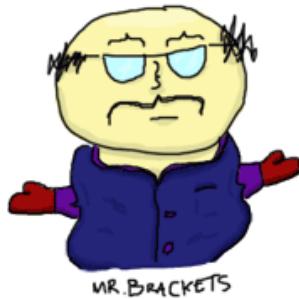
```
1> X = 10, Y = 4.  
4  
2> Point = {X,Y}.  
{10,4}
```

In this case, a point will always be two terms. Instead of carrying the variables X and Y around the place, you only have to carry one instead. However, what can I do if I receive a point and only want the X coordinate? It's not hard to extract that information. Remember that when we assigned values, Erlang would never complain if they were the same. Let's exploit that! You may need to clean the variables we had set with f().

```
3> Point = {4,5}.  
{4,5}  
4> {X,Y} = Point.  
{4,5}  
5> X.  
4  
6> {X,_} = Point.  
{4,5}
```

From then on we can use X to get the first value of the tuple! How did that happen? First, X and Y had no value and were thus considered unbound

variables. When we set them in the tuple $\{X,Y\}$ on the left-hand side of the `=` operator, the `=` operator compares both values: $\{X,Y\}$ vs. $\{4,5\}$. Erlang is smart enough to unpack the values from the tuple and distribute them to the unbound variables on the left-hand side. Then the comparison is only $\{4,5\} = \{4,5\}$, which obviously succeeds! That's one of the many forms of pattern matching.



Note that on expression 6, I used the anonymous `_` variable. This is exactly how it's meant to be used: to drop the value that would usually be placed there since we won't use it. The `_` variable is always seen as unbound and acts as a wildcard for pattern matching. Pattern matching to unpack tuples will only work if the number of elements (the tuple's length) is the same.

```
7> {_,_} = {4,5}.  
{4,5}  
8> {_,_} = {4,5,6}.  
** exception error: no match of right hand side value {4,5,6}
```

Tuples can also be useful when working with single values. How so? The simplest example is temperature:

```
9> Temperature = 23.213.  
23.213
```

Well, it sounds like a good day to go to the beach... Wait, is this temperature in Kelvin, Celsius or Fahrenheit?

```
10> PreciseTemperature = {celsius, 23.213}.  
{celsius,23.213}
```

```
11> {kelvin, T} = PreciseTemperature.  
** exception error: no match of right hand side value {celsius,23.213}
```

This throws an error, but it's exactly what we want! This is, again, pattern matching at work. The `=` operator ends up comparing `{kelvin, T}` and `{celsius, 23.213}`: even if the variable `T` is unbound, Erlang won't see the `celsius` atom as identical to the `kelvin` atom when comparing them. An exception is thrown which stops the execution of code. By doing so, the part of our program that expects a temperature in Kelvin won't be able to process temperatures sent in Celsius. This makes it easier for the programmer to know what is being sent around and also works as a debugging aid. A tuple which contains an atom with one element following it is called a 'tagged tuple'. Any element of a tuple can be of any type, even another tuple:

```
12> {point, {X,Y}}.  
{point,{4,5}}
```

What if we want to carry around more than one Point though?

Lists!

Lists are the bread and butter of many functional languages. They're used to solve all kinds of problems and are undoubtedly the most used data structure in Erlang. Lists can contain anything! Numbers, atoms, tuples, other lists; your wildest dreams in a single structure. The basic notation of a list is `[Element1, Element2, ..., ElementN]` and you can mix more than one type of data in it:

```
1> [1, 2, 3, {numbers,[4,5,6]}, 5.34, atom].  
[1,2,3,{numbers,[4,5,6]},5.34,atom]
```

Simple enough, right?

```
2> [97, 98, 99].  
"abc"
```

Uh oh! This is one of the most disliked things in Erlang: strings! Strings are lists and the notation is absolutely the exact same! Why do people dislike it? Because of this:

```
3> [97,98,99,4,5,6].  
[97,98,99,4,5,6]  
4> [233].  
"é"
```

Erlang will print lists of numbers as numbers only when at least one of them could not also represent a letter! There is no such thing as a real string in Erlang! This will no doubt come to haunt you in the future and you'll hate the language for it. Don't despair, because there are other ways to write strings we'll see later in this chapter.

Don't drink too much Kool-Aid:

This is why you may have heard Erlang is said to suck at string manipulation: there is no built-in string type like in most other languages. This is because of Erlang's origins as a language created and used by telecom companies. They never (or rarely) used strings and as such, never felt like adding them officially. However, most of Erlang's lack of sense in string manipulations is getting fixed with time: The VM now natively supports Unicode strings, and overall gets faster on string manipulations all the time.

There is also a way to store strings as a binary data structure, making them really light and faster to work with. All in all, there are still some functions missing from the standard library and while string processing is definitely doable in Erlang, there are somewhat better languages for tasks that need lots of it, like Perl or Python.

To glue lists together, we use the `++` operator. The opposite of `++` is `--` and will remove elements from a list:

```
5> [1,2,3] ++ [4,5].  
[1,2,3,4,5]  
6> [1,2,3,4,5] -- [1,2,3].
```

```
[4,5]  
7> [2,4,2] -- [2,4].  
[2]  
8> [2,4,2] -- [2,4,2].  
[]
```

Both ++ and -- are right-associative. This means the elements of many -- or ++ operations will be done from right to left, as in the following examples:

```
9> [1,2,3] -- [1,2] -- [3].  
[3]  
10> [1,2,3] -- [1,2] -- [2].  
[2,3]
```

Let's keep going. The first element of a list is named the Head, and the rest of the list is named the Tail. We will use two built-in functions (BIF) to get them.

```
11> hd([1,2,3,4]).  
1  
12> tl([1,2,3,4]).  
[2,3,4]
```

Note: built-in functions (BIFs) are usually functions that could not be implemented in pure Erlang, and as such are defined in C, or whichever language Erlang happens to be implemented on (it was Prolog in the 80's). There are still some BIFs that could be done in Erlang but were still implemented in C in order to provide more speed to common operations. One example of this is the `length(List)` function, which will return the (you've guessed it) length of the list passed in as the argument.

Accessing or adding the head is fast and efficient: virtually all applications where you need to deal with lists will always operate on the head first. As it's used so frequently, there is a nicer way to separate the head from the tail of a list with the help of pattern matching: [Head|Tail]. Here's how you would add a new head to a list:

```
13> List = [2,3,4].  
[2,3,4]  
14> NewList = [1|List].  
[1,2,3,4]
```

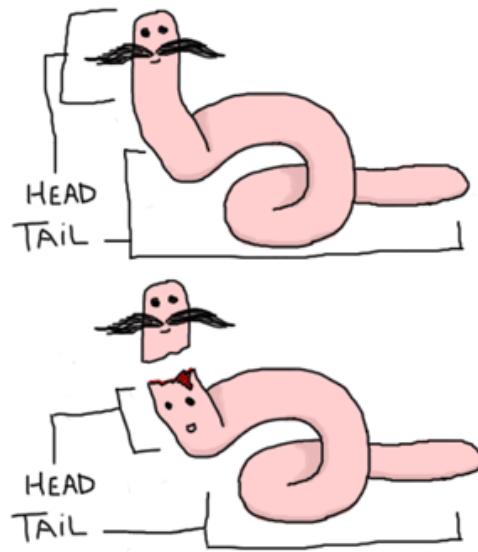
When processing lists, as you usually start with the head, you want a quick way to also store the tail to later operate on it. If you remember the way tuples work and how we used pattern matching to unpack the values of a point ($\{X,Y\}$), you'll know we can get the first element (the head) sliced off a list in a similar manner.

```
15> [Head|Tail] = NewList.  
[1,2,3,4]  
16> Head.  
1  
17> Tail.  
[2,3,4]  
18> [NewHead|NewTail] = Tail.  
[2,3,4]  
19> NewHead.  
2
```

The `|` we used is named the cons operator (constructor). In fact, any list can be built with only cons and values:

```
20> [1|[]].  
[1]  
21> [2|[1|[]]].  
[2,1]  
22> [3|[2|[1|[]]]].  
[3,2,1]
```

This is to say any list can be built with the following formula: `[Term1| [Term2 | ... | [TermN]]]]`.... Lists can thus be defined recursively as a head preceding a tail, which is itself a head followed by more heads. In this sense we could imagine a list being a bit like an earthworm: you can slice it in half and you'll then have two worms.



The ways Erlang lists can be built are sometimes confusing to people who are not used to similar constructors. To help you get familiar with the concept, read all of these examples (hint: they're all equivalent):

```
[a, b, c, d]
[a, b, c, d | []]
[a, b | [c, d]]
[a, b | [c | [d]]]
[a | [b | [c | [d]]]]
[a | [b | [c | [d | [] ]]]]
```

With this understood, you should be able to deal with list comprehensions.

Note: Using the form `[1|2]` gives what we call an 'improper list'. Improper lists will work when you pattern match in the `[Head|Tail]` manner, but will fail to be used with standard functions of Erlang (even `length()`). This is because Erlang expects proper lists. Proper lists end with an empty list as their last cell. When declaring an item like `[2]`, the list is automatically formed in a proper manner. As such, `[1|[2]]` would work! Improper lists, although syntactically valid, are of very limited use outside of user-defined data structures.

List Comprehensions

List comprehensions are ways to build or modify lists. They also make programs short and easy to understand compared to other ways of manipulating lists. It's based off the idea of set notation; if you've ever taken mathematics classes with set theory or if you've ever looked at mathematical notation, you probably know how that works. Set notation basically tells you how to build a set by specifying properties its members must satisfy. List comprehensions may be hard to grasp at first, but they're worth the effort. They make code cleaner and shorter, so don't hesitate to try and type in the examples until you understand them!

An example of set notation would be $\{x \in \mathbf{R} : x = x^2\}$. That set notation tells you the results you want will be all real numbers who are equal to their own square. The result of that set would be $\{0,1\}$. Another set notation example, simpler and abbreviated would be $\{x : x > 0\}$. Here, what we want is all numbers where $x > 0$.

List comprehensions in Erlang are about building sets from other sets. Given the set $\{2n : n \text{ in } L\}$ where L is the list $[1,2,3,4]$, the Erlang implementation would be:

```
1> [2*N || N <- [1,2,3,4]].  
[2,4,6,8]
```

Compare the mathematical notation to the Erlang one and there's not a lot that changes: brackets ($\{\}$) become square brackets ($[]$), the colon ($:$) becomes two pipes ($||$) and the word 'in' becomes the arrow ($<-$). We only change symbols and keep the same logic. In the example above, each value of $[1,2,3,4]$ is sequentially pattern matched to N . The arrow acts exactly like the $=$ operator, with the exception that it doesn't throw exceptions.

You can also add constraints to a list comprehension by using operations that return boolean values. If we wanted all the even numbers from one to ten, we could write something like:

```
2> [X || X <- [1,2,3,4,5,6,7,8,9,10], X rem 2 == 0].  
[2,4,6,8,10]
```

Where $x \text{ rem } 2 == 0$ checks if a number is even. Practical applications come when we decide we want to apply a function to each element of a list, forcing it to respect constraints, etc. As an example, say we own a restaurant. A customer enters, sees our menu and asks if he could have the prices of all the items costing between \$3 and \$10 with taxes (say 7%) counted in afterwards.

```
3> RestaurantMenu = [{steak, 5.99}, {beer, 3.99}, {poutine, 3.50}, {kitten, 20.99}, {water, 0.00}].  
[{steak,5.99},  
{beer,3.99},  
{poutine,3.5},  
{kitten,20.99},  
{water,0.0}]  
4> [{Item, Price*1.07} || {Item, Price} <- RestaurantMenu, Price >= 3, Price =< 10].  
[{steak,6.409300000000001},{beer,4.2693},{poutine,3.745}]
```

Of course, the decimals aren't rounded in a readable manner, but you get the point. The recipe for list comprehensions in Erlang is therefore NewList = [Expression || Pattern <- List, Condition1, Condition2, ... ConditionN]. The part Pattern <- List is named a Generator expression. You can have more than one!

```
5> [X+Y || X <- [1,2], Y <- [2,3]].  
[3,4,4,5]
```

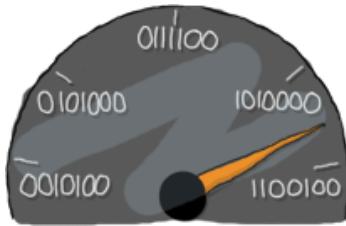
This runs the operations 1+2, 1+3, 2+2, 2+3. So if you want to make the list comprehension recipe more generic, you get: NewList = [Expression || GeneratorExp1, GeneratorExp2, ..., GeneratorExpN, Condition1, Condition2, ... ConditionM]. Note that the generator expressions coupled with pattern matching also act as a filter:

```
6> Weather = [{toronto, rain}, {montreal, storms}, {london, fog},  
6>           {paris, sun}, {boston, fog}, {vancouver, snow}].  
[{toronto,rain},  
{montreal,storms},  
{london,fog},  
{paris,sun},  
{boston,fog},  
{vancouver,snow}]
```

```
7> FoggyPlaces = [X || {X, fog} <- Weather].  
[london,boston]
```

If an element of the list 'Weather' doesn't match the {X, fog} pattern, it's simply ignored in the list comprehension whereas the = operator would have thrown an exception.

There is one more basic data type left for us to see for now. It is a surprising feature that makes interpreting binary data easy as pie.



Bit Syntax!

Most languages have support for manipulating data such as numbers, atoms, tuples, lists, records and/or structs, etc. Most of them also only have very raw facilities to manipulate binary data. Erlang goes out of its way to provide useful abstractions when dealing with binary values with pattern matching taken to the next level. It makes dealing with raw binary data fun and easy (no, really), which was necessary for the telecom applications it was created to help with. Bit manipulation has a unique syntax and idioms that may look kind of weird at first, but if you know how bits and bytes generally work, this should make sense to you.

You may want to skip the rest of this chapter otherwise.

Bit syntax encloses binary data between << and >>, splits it in readable segments, and each segment is separated by a comma. A segment is a sequence of bits of a binary (not necessarily on a byte boundary, although this is the default behaviour). Say we want to store an orange pixel of true color (24 bits). If you've ever checked colors in Photoshop or in a CSS style sheet for the web, you know the hexadecimal notation has the format #RRGGBB. A tint of orange is #F09A29 in that notation, which could be expanded in Erlang to:

```
1> Color = 16#F09A29.  
15768105  
2> Pixel = <<Color:24>>.  
<<240,154,41>>
```

This basically says "Put the binary values of `#F09A29` on 24 bits of space (Red on 8 bits, Green on 8 bits and Blue also on 8 bits) in the variable `Pixel`." The value can later be taken to be written to a file. This doesn't look like much, but once written to a file, what you'd get by opening it in a text editor would be a bunch of unreadable characters. When you read back from the file, Erlang would interpret the binary into the nice `<<240,151,41>>` format again!

What's more interesting is the ability to pattern match with binaries to unpack content:

```
3> Pixels = <<213,45,132,64,76,32,76,0,0,234,32,15>>.  
<<213,45,132,64,76,32,76,0,0,234,32,15>>  
4> <<Pixel1,Pixel2,Pixel3,Pixel4>> = Pixels.  
** exception error: no match of right hand side value <<213,45,132,64,76,32,76,  
0,0,234,32,15>>  
5> <<Pixel1:24, Pixel2:24, Pixel3:24, Pixel4:24>> = Pixels.  
<<213,45,132,64,76,32,76,0,0,234,32,15>>
```

What we did on command 3 is declare what would be precisely 4 pixels of RGB colors in binary.

On expression 4, we tried to unpack 4 values from the binary content. It throws an exception, because we have more than 4 segments, we in fact have 12! So what we do is tell Erlang that each variable on the left side will hold 24 bits of data. That's what `Var:24` means. We can then take the first pixel and unpack it further into single color values:

```
6> <<R:8, G:8, B:8>> = <<Pixel1:24>>.  
<<213,45,132>>  
7> R.  
213
```

"Yeah that's dandy. What if I only wanted the first color from the start though? will I have to unpack all these values all the time?" Hah! Doubt not! Erlang introduces more syntactic sugar and pattern matching to help you around:

```
8> <<R:8, Rest/binary>> = Pixels.  
<<213,45,132,64,76,32,76,0,0,234,32,15>>  
9> R.  
213
```

Nice, huh? That's because Erlang accepts more than one way to describe a binary segment. Those are all valid:

Value
Value:Size
Value/TypeSpecifierList
Value:Size/TypeSpecifierList

where *Size* is going to represent bits or bytes (depending on *Type* and *Unit* below), and *TypeSpecifierList* represents one or more of the following:

Type

Possible values: integer | float | binary | bytes | bitstring | bits | utf8 | utf16 | utf32
This represents the kind of binary data used. Note that 'bytes' is shorthand for 'binary' and 'bits' is shorthand for 'bitstring'. When no type is specified, Erlang assumes an 'integer' type.

Signedness

Possible values: signed | unsigned
Only matters for matching when the type is integer. The default is 'unsigned'.

Endianness

Possible values: big | little | native
Endianness only matters when the Type is either integer, utf16, utf32, or float. This has to do with how the system reads binary data. As an example, the BMP image header format holds the size of its file as an integer stored on 4 bytes. For a file that has a size of 72 bytes, a little-

endian system would represent this as <<72,0,0,0>> and a big-endian one as <<0,0,0,72>>. One will be read as '72' while the other will be read as '1207959552', so make sure you use the right endianness. There is also the option to use 'native', which will choose at run-time if the CPU uses little-endianness or big-endianness natively. By default, endianness is set to 'big'.

Unit

written unit:Integer

This is the size of each segment, in bits. The allowed range is 1..256 and is set by default to 1 for integers, floats and bit strings and to 8 for binary. The utf8, utf16 and utf32 types require no unit to be defined. The multiplication of Size by Unit is equal to the number of bits the segment will take and must be evenly divisible by 8. The unit size is usually used to ensure byte-alignment.

The TypeSpecifierList is built by separating attributes by a '-'.

Some examples may help digest the definitions:

```
10> <<X1/unsigned>> = <<-44>>.  
<<"Ô">>  
11> X1.  
212  
12> <<X2/signed>> = <<-44>>.  
<<"Ô">>  
13> X2.  
-44  
14> <<X2/integer-signed-little>> = <<-44>>.  
<<"Ô">>  
15> X2.  
-44  
16> <<N:8/unit:1>> = <<72>>.  
<<"H">>  
17> N.  
72  
18> <<N/integer>> = <<72>>.  
<<"H">>  
19> <<Y:4/little-unit:8>> = <<72,0,0,0>>.
```

```
<<72,0,0,0>>
20> Y.
72
```

You can see there are more than one way to read, store and interpret binary data. This is a bit confusing, but still much simpler than using the usual tools given by most languages.

The standard binary operations (shifting bits to left and right, binary 'and', 'or', 'xor', or 'not') also exist in Erlang. Just use the functions `bsl` (Bit Shift Left), `bsr` (Bit Shift Right), `band`, `bor`, `bxor`, and `bnot`.

```
2#00100 = 2#00010 bsl 1.
2#00001 = 2#00010 bsr 1.
2#10101 = 2#10001 bor 2#00101.
```

With that kind of notation and the bit syntax in general, parsing and pattern matching binary data is a piece of cake. One could parse TCP segments with code like this:

```
<<SourcePort:16, DestinationPort:16,
AckNumber:32,
DataOffset:4, _Reserved:4, Flags:8, WindowSize:16,
CheckSum: 16, UrgentPointer:16,
Payload/binary>> = SomeBinary.
```

The same logic can then be applied to anything binary: video encoding, images, other protocol implementations, etc.

Don't drink too much Kool-Aid:

Erlang is slow compared to languages like C or C++. Unless you are a patient person, it would be a bad idea to do stuff like converting videos or images with it, even though the binary syntax makes it extremely interesting as I hinted above. Erlang is just not that great at heavy number crunching.

Take note, however, that Erlang is still mighty fast for applications that do not require number crunching: reacting to events, message passing

(with the help of atoms being extremely light), etc. It can deal with events in matters of milliseconds and as such is a great candidate for soft-real-time applications.



There's a whole other aspect to binary notation: bit strings. Binary strings are bolted on top of the language the same way they are with lists, but they're much more efficient in terms of space. This is because normal lists are linked lists (1 'node' per letter) while bit strings are more like C arrays. Bit strings use the syntax `<<"this is a bit string!">>`. The downside of binary strings compared to lists is a loss in simplicity when it comes to pattern matching and manipulation. Consequently, people tend to use binary strings when storing text that won't be manipulated too much or when space efficiency is a real issue.

Note: Even though bit strings are pretty light, you should avoid using them to tag values. It could be tempting to use string literals to say `{<<"temperature">>,50}`, but always use atoms when doing that. Previously in this chapter, atoms were said to be taking only 4 or 8 bytes in space, no matter how long they are. By using them, you'll have basically no overhead when copying data from function to function or sending it to another Erlang node on another server.

Conversely, do not use atoms to replace strings because they are lighter. Strings can be manipulated (splitting, regular expressions, etc) while atoms can only be compared and nothing else.

Binary Comprehensions

Binary comprehensions are to bit syntax what list comprehensions are to lists: a way to make code short and concise. They are relatively new in the Erlang world as they were there in previous revisions of Erlang, but

required a module implementing them to use a special compile flag in order to work. Since the R13B revisions (those used here), they've become standard and can be used anywhere, including the shell:

```
1> [ X || <<X>> <= <<1,2,3,4,5>>, X rem 2 == 0 ].  
[2,4]
```

The only change in syntax from regular list comprehensions is the `<-` which became `<=` and using binaries (`<<>>`) instead of lists (`[]`). Earlier in this chapter we've seen an example where there was a binary value of many pixels on which we used pattern matching to grab the RGB values of each pixel. It was alright, but on larger structures, it would become possibly harder to read and maintain. The same exercise can be done with a one-line binary comprehension, which is much cleaner:

```
2> Pixels = <<213,45,132,64,76,32,76,0,0,234,32,15>>.  
<<213,45,132,64,76,32,76,0,0,234,32,15>>  
3> RGB = [ {R,G,B} || <<R:8,G:8,B:8>> <= Pixels ].  
[{213,45,132},{64,76,32},{76,0,0},{234,32,15}]
```

Changing `<-` to `<=` let us use a binary stream as a generator. The complete binary comprehension basically changed binary data to integers inside tuples. Another binary comprehension syntax exists to let you do the exact opposite:

```
4> <<<R:8, G:8, B:8>> || {R,G,B} <- RGB >>.  
<<213,45,132,64,76,32,76,0,0,234,32,15>>
```

Be careful, as the elements of the resulting binary require a clearly defined size if the generator returned binaries:

```
5> <<<Bin>> || Bin <- [<<3,7,5,4,7>>] >>.  
** exception error: bad argument  
6> <<<Bin/binary>> || Bin <- [<<3,7,5,4,7>>] >>.  
<<3,7,5,4,7>>
```

It's also possible to have a binary comprehension with a binary generator, given the fixed-size rule above is respected:

```
7> <<<(x+1)/integer>> || <<x>> <= <<3,7,5,4,7>> >>.  
<<4,8,6,5,8>>
```

Note: At the time of this writing, binary comprehensions were seldom used and not documented very well. As such, it was decided not to dig more than what is necessary to identify them and understand their basic working. To understand more bit syntax as a whole, read the white paper defining their specification.

Modules

What are modules



Working with the interactive shell is often considered a vital part of using dynamic programming languages. It is useful to test all kinds of code and programs. Most of the basic data types of Erlang were used without even needing to open a text editor or saving files. You could drop your keyboard, go play ball outside and call it a day, but you would be a terrible Erlang programmer if you stopped right there. Code needs to be saved somewhere to be used!

This is what modules are for. Modules are a bunch of functions regrouped in a single file, under a single name. Additionally, all functions in Erlang must be defined in modules. You have already used modules, perhaps without realizing it. The BIFs mentioned in the previous chapter, like `hd` or `tl`, actually belong to the `erlang` module, as well as all of the arithmetic, logic and Boolean operators. BIFs from the `erlang` module differ from other functions as they are automatically imported when you use Erlang. Every other function defined

in a module you will ever use needs to be called with the form `Module:Function(Arguments)`.

You can see for yourself:

```
1> erlang:element(2, {a,b,c}).  
b  
2> element(2, {a,b,c}).  
b  
3> lists:seq(1,4).  
[1,2,3,4]  
4> seq(1,4).  
** exception error: undefined shell command seq/2
```

Here, the `seq` function from the `list` module was not automatically imported, while `element` was. The error 'undefined shell command' comes from the shell looking for a shell command like `f()` and not being able to find it. There are some functions from the `erlang` module which are not automatically imported, but they're not used too frequently.

Logically, you should put functions about similar things inside a single module. Common operations on lists are kept in the `lists` module, while functions to do input and output (such as writing to the terminal or in a file) are regrouped in the `io` module. One of the only modules you will encounter which doesn't respect that pattern is the aforementioned `erlang` module that has functions which do math, conversions, deal with multiprocessing, fiddle with the virtual machine's settings, etc. They have no point in common except being

built-in functions. You should avoid creating modules like `erlang` and instead focus on clean logical separations.

Module Declaration



When writing a module, you can declare two kinds of things: *functions* and *attributes*. Attributes are metadata describing the module itself such as its name, the functions that should be visible to the outside world, the author of the code, and so on. This kind of metadata is useful because it gives hints to the compiler on how it should do its job, and also because it lets people retrieve useful information from compiled code without having to consult the source.

There is a large variety of module attributes currently used in Erlang code across the world; as a matter of fact, you can even declare your own attributes for whatever you please. There are some pre-defined attributes that will appear more frequently than others in your code. All module attributes follow the form `-Name(Attribute)`. Only one of them is necessary for your module to be compilable:

`-module(Name).`

This is always the first attribute (and statement) of a file, and for good reason: it's the name of the current module, where *Name* is an [atom](#). This is the name you'll use to call functions from other modules. The calls are made with the $M:F(A)$ form, where *M* is the module name, *F* the function, and *A* the arguments.

It's time to code already! Our first module will be very simple and useless. Open your text editor and type in the following, then save it under `useless.erl`:

```
-module(useless).
```

This line of text is a valid module. Really! Of course it's useless without functions. Let's first decide what functions will be exported from our 'useless' module. To do this, we will use another attribute:

```
-export([Function1/Arity, Function2/Arity, ..., FunctionN/Arity]).
```

This is used to define what functions of a module can be called by the outside world. It takes a list of functions with their respective arity. The arity of a function is an integer representing how many arguments can be passed to the function. This is critical information, because different functions defined within a module can share the same name if and only if they have a different arity. The functions `add(X,Y)` and `add(X,Y,Z)` would thus be considered different and written in the form `add/2` and `add/3` respectively.

Note: Exported functions represent a module's interface. It is important to define an interface revealing strictly what is necessary for it to be used and nothing more. Doing so lets you fiddle with all the other [hidden] details of your implementation without breaking code that might depend on your module.

Our useless module will first export a useful function named 'add', which will take two arguments. The following `-export` attribute can be added after the module declaration:

```
-export([add/2]).
```

And now write the function:

```
add(A,B) ->  
    A + B.
```

The syntax of a function follows the form `Name(Args) -> Body,`, where *Name* has to be an atom and *Body* can be one or more Erlang expressions separated by commas. The function is ended with a period. Note that Erlang doesn't use the 'return' keyword. 'Return' is useless! Instead, the last logical expression of a function to be executed will have its value returned to the caller automatically without you having to mention it.

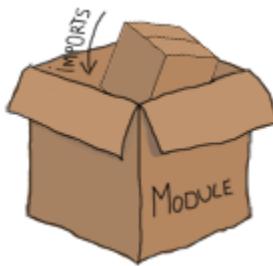
Add the following function (why yes, every tutorial needs a 'Hello world' example! Even at the fourth chapter!), without forgetting to add it to the `-export` attribute.

```
%% Shows greetings.  
%% io:format/1 is the standard function used to output text.  
hello() ->  
    io:format("Hello, world!~n").
```

What we see from this function is that comments are single-line only and begin with a % sign (using %% is purely a question of style.) The hello/0 function also demonstrates how to call functions from foreign modules inside yours. In this case, io:format/1 is the standard function to output text, as written in the comments.

A last function will be added to the module, using both functions add/2 and hello/0:

```
greet_and_add_two(X) ->  
    hello(),  
    add(X,2).
```



Do not forget to add greet_and_add_two/1 to the exported function list. The calls to hello/0 and add/2 don't need to have the module name prepended to them because they were declared in the module itself.

Had you wanted to be able to call io:format/1 in the same manner as add/2 or any other function defined within the

module, you could have added the following module attribute at the beginning of the file: `-import(io, [format/1]).` Then you could have called `format("Hello, World!~n")`. directly. More generally, the `-import` attribute follows this recipe:

```
-import(Module, [Function1/Arity, ..., FunctionN/Arity]).
```

Importing a function is not much more than a shortcut for programmers when writing their code. Erlang programmers are often discouraged from using the `-import` attribute as some people find it reduces the readability of code. In the case of `io:format/2`, the function `io_lib:format/2` also exists.

Finding which one is used means going to the top of the file to see from which module it was imported. Consequently, leaving the module name in is considered good practice. Usually, the only functions you'll see imported come from the lists module: its functions are used with a higher frequency than those from most other modules.

Your useless module should now look like the following file:

```
-module(useless).  
-export([add/2, hello/0, greet_and_add_two/1]).
```

```
add(A,B) ->  
    A + B.
```

```
%% Shows greetings.  
%% io:format/1 is the standard function used to output text.  
hello() ->  
    io:format("Hello, world!~n").
```

```
greet_and_add_two(X) ->
    hello(),
    add(X,2).
```

We are done with the "useless" module. You can save the file under the name `useless.erl`. The file name should be the module name as defined in the `-module` attribute, followed by `'.erl'`, which is the standard Erlang source extension.

Before showing how to compile the module and finally try all its exciting functions, we will see how to define and use macros. Erlang macros are really similar to C's `#define` statements, mainly used to define short functions and constants. They are simple expressions represented by text that will be replaced before the code is compiled for the VM. Such macros are mainly useful to avoid having magic values floating around your modules. A macro is defined as a module attribute of the form: `-define(MACRO, some_value)`. and is used as `?MACRO` inside any function defined in the module. A 'function' macro could be written as `-define(sub(X,Y), X-Y)`. and used like `?sub(23,47)`, later replaced by `23-47` by the compiler. Some people will use more complex macros, but the basic syntax stays the same.

Compiling the code

Erlang code is compiled to bytecode in order to be used by the virtual machine. You can call the compiler from many places: `$ erlc flags file.erl` when in the command line,

`compile:file(File)` when in the shell or in a module, `c()` when in the shell, etc.

It's time to compile our useless module and try it. Open the Erlang shell, type in:

```
1> cd("/path/to/where/you/saved/the-module/").  
"Path Name to the directory you are in"  
ok
```

By default, the shell will only look for files in the same directory it was started in and the standard library: `cd/1` is a function defined exclusively for the Erlang shell, telling it to change the directory to a new one so it's less annoying to browse for our files. Windows users should remember to use forward slashes. When this is done, do the following:

```
2> c(useless).  
{ok,useless}
```

If you have another message, make sure the file is named correctly, that you are in the right directory and that you've made no mistake in your module. Once you successfully compile code, you'll notice that a `useless.beam` file was added next to `useless.erl` in your directory. This is the compiled module. Let's try our first functions ever:

```
3> useless:add(7,2).  
9  
4> useless:hello().  
Hello, world!  
ok
```

```
5> useless:greet_and_add_two(-3).
Hello, world!
-1
6> useless:not_a_real_function().
** exception error: undefined function useless:not_a_real_function/0
```

The functions work as expected: `add/2` adds numbers, `hello/0` outputs "Hello, world!", and `greet_and_add_two/1` does both! Of course, you might be asking why `hello/0` returns the atom 'ok' after outputting text. This is because Erlang functions and expressions must **always** return something, even if they would not need to in other languages. As such, `io:format/1` returns 'ok' to denote a normal condition, the absence of errors.

Expression 6 shows an error being thrown because a function doesn't exist. If you have forgotten to export a function, this is the kind of error message you will have when trying it out.

Note: If you were ever wondering, '.beam' stands for *Bogdan/Björn's Erlang Abstract Machine*, which is the VM itself. Other virtual machines for Erlang exist, but they're not really used anymore and are history: JAM (Joe's Abstract Machine, inspired by Prolog's WAM and old BEAM, which attempted to compile Erlang to C, then to native code). Benchmarks demonstrated little benefits in this practice and the concept was given up.

There are a whole lot of compilation flags existing to get more control over how a module is compiled. You can get a list of all of them in the Erlang documentation. The most common flags are:

-debug_info

Erlang tools such as debuggers, code coverage and static analysis tools will use the debug information of a module in order to do their work.

-{outdir,Dir}

By default, the Erlang compiler will create the 'beam' files in the current directory. This will let you choose where to put the compiled file.

-export_all

Will ignore the -export module attribute and will instead export all functions defined. This is mainly useful when testing and developing new code, but should not be used in production.

-{d,Macro} or {d,Macro,Value}

Defines a macro to be used in the module, where *Macro* is an atom. This is more frequently used when dealing with unit-testing, ensuring that a module will only have its testing functions created and exported when they are explicitly wanted. By default, *Value* is 'true' if it's not defined as the third element of the tuple.

To compile our useless module with some flags, we could do one of the following:

```
7> compile:file(useless, [debug_info, export_all]).  
{ok,useless}  
8> c(useless, [debug_info, export_all]).  
{ok,useless}
```

You can also be sneaky and define compile flags from within a module, with a module attribute. To get the same results as from expressions 7 and 8, the following line could be added to the module:

```
-compile([debug_info, export_all]).
```

Then just compile and you'll get the same results as if you manually passed flags. Now that we're able to write down functions, compile them and execute them, it's time to see how far we can take them!

Note: another option is to compile your Erlang module to native code. Native code compiling is **not** available for every platform and OS, but on those that support it, it can make your programs go faster (about 20% faster, based on anecdotal evidence). To compile to native code, you need to use the `hipe` module and call it the following way:

`hipe:c(Module,OptionsList)`. You could also use `c(Module,[native])`, when in the shell to achieve similar results. Note that the .beam file generated will contain both native and non-native code, and the native part will not be portable across platforms.

More About Modules

Before moving on to learning more about writing functions and barely useful snippets of code, there are a few other miscellaneous bits of information that might be useful to you in the future that I'd like to discuss.

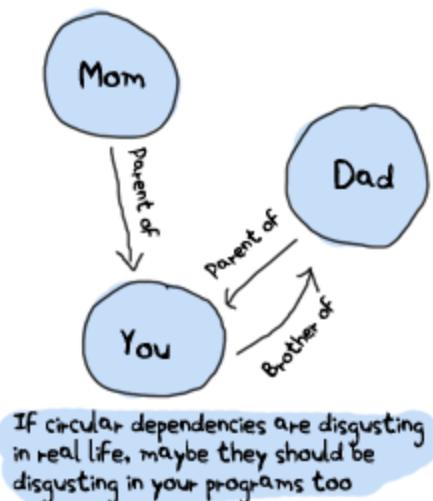
The first one concerns metadata about modules. I mentioned in the beginning of this chapter that module attributes are metadata describing the module itself. Where can we find this metadata when we don't have an access to the source? Well the compiler plays nice with us: when compiling a module, it will pick up most module attributes and store them (along with other information) in a `module_info/0` function. You can see the metadata of the useless module the following way:

```
9> useless:module_info().  
[{exports,[{add,2},  
          {hello,0},  
          {greet_and_add_two,1},  
          {module_info,0},  
          {module_info,1}]],  
 {imports,[]},  
 {attributes,[{vsn,[174839656007867314473085021121413256129]}]},  
 {compile,[{options,[]},  
          {version,"4.6.2"},  
          {time,{2009,9,9,22,15,50}},  
          {source,"/home/ferd/learn-you-some-erlang/useless.erl"}]]  
10> useless:module_info(attributes).  
[{vsn,[174839656007867314473085021121413256129}]}
```

The snippet above also shows an additional function, `module_info/1` which will let you grab one specific piece of

information. You can see exported functions, imported functions (none in this case!), attributes (this is where your custom metadata would go), and compile options and information. Had you decided to add `-author("An Erlang Champ")`. to your module, it would have ended up in the same section as `vsn`. There are limited uses to module attributes when it comes to production stuff, but they can be nice when doing little tricks to help yourself out: I'm using them in my testing script for this book to annotate functions for which unit tests could be better; the script looks up module attributes, finds the annotated functions and shows a warning about them.

Note: `vsn` is an automatically generated unique value differentiating each version of your code, excluding comments. It is used in code hot-loading (upgrading an application while it runs, without stopping it) and by some tools related to release handling. You can also specify a `vsn` value yourself if you want: just add `-vsn(VersionNumber)` to your module.



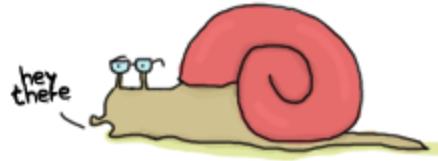
Another point that would be nice to approach regards general module design: avoid circular dependencies! A module *A* should not call a module *B* that also calls module *A*. Such dependencies usually end up making code maintenance difficult. In fact, depending on too many modules even if they're not in a circular dependency can make maintenance harder. The last thing you want is to wake up in the middle of the night only to find a maniac software engineer or computer scientist trying to gouge your eyes out because of terrible code you have written.

For similar reasons (maintenance and fear for your eyes), it is usually considered a good practice to regroup functions that have similar roles close together. Starting and stopping an application or creating and deleting a record in some database are examples of such a scenario.

Well, that's enough for the pedantic moralizations. How about we explore Erlang a little more?

Syntax in functions

Pattern Matching



Now that we have the ability to store and compile our code, we can begin to write more advanced functions. Those that we have written so far were extremely simple and a bit underwhelming. We'll get to more interesting stuff. The first function we'll write will need to greet someone differently according to gender. In most languages you would need to write something similar to this:

```
function greet(Gender,Name)
    if Gender == male then
        print("Hello, Mr. %s!", Name)
    else if Gender == female then
        print("Hello, Mrs. %s!", Name)
    else
        print("Hello, %s!", Name)
end
```

With pattern-matching, Erlang saves you a whole lot of boilerplate code. A similar function in Erlang would look like this:

```
greet(male, Name) ->
    io:format("Hello, Mr. ~s!", [Name]);
```

```
greet(female, Name) ->
    io:format("Hello, Mrs. ~s!", [Name]);
greet(_, Name) ->
    io:format("Hello, ~s!", [Name]).
```

I'll admit that the printing function is a lot uglier in Erlang than in many other languages, but that is not the point. The main difference here is that we used pattern matching to define both what parts of a function should be used and bind the values we need at the same time. There was no need to first bind the values and then compare them! So instead of:

```
function(Args)
if X then
    Expression
else if Y then
    Expression
else
    Expression
```

We write:

```
function(x) ->
    Expression;
function(y) ->
    Expression;
function(_) ->
    Expression.
```

in order to get similar results, but in a much more declarative style. Each of these function declarations is called a *function clause*. Function clauses must be separated by semicolons

(;) and together form a *function declaration*. A function declaration counts as one larger statement, and it's why the final function clause ends with a period. It's a "funny" use of tokens to determine workflow, but you'll get used to it. At least you'd better hope so because there's no way out of it!

Note: io:format's formatting is done with the help of tokens being replaced in a string. The character used to denote a token is the tilde (~). Some tokens are built-in such as ~n, which will be changed to a line-break. Most other tokens denote a way to format data. The function call io:format("~s!~n", ["Hello"]). includes the token ~s, which accepts strings and bitstrings as arguments, and ~n. The final output message would thus be "Hello!\n". Another widely used token is ~p, which will print an Erlang term in a nice way (adding in indentation and everything).

The io:format function will be seen in more details in later chapters dealing with input/output with more depth, but in the meantime you can try the following calls to see what they do: io:format(~s~n,[<<"Hello">>]), io:format(~p~n,[<<"Hello">>]), io:format(~~~n), io:format(~f~n, [4.0]), io:format(~30f~n, [4.0]). They're a small part of all that's possible and all in all they look a bit like printf in many other languages. If you can't wait until the chapter about I/O, you can read the online documentation to know more.

Pattern matching in functions can get more complex and powerful than that. As you may or may not remember from a

few chapters ago, we can pattern match on lists to get the heads and tails. Let's do this! Start a new module called functions in which we'll write a bunch of functions to explore many pattern matching avenues available to us:

```
-module(functions).  
-compile(export_all). %% replace with -export() later, for God's sake!
```

The first function we'll write is head/1, acting exactly like erlang:hd/1 which takes a list as an argument and returns its first element. It'll be done with the help of the cons operator (|):

```
head([H|_]) -> H.
```

If you type functions:head([1,2,3,4]). in the shell (once the module is compiled), you can expect the value '1' to be given back to you. Consequently, to get the second element of a list you would create the function:

```
second([_,X|_]) -> X.
```

The list will just be deconstructed by Erlang in order to be pattern matched. Try it in the shell!

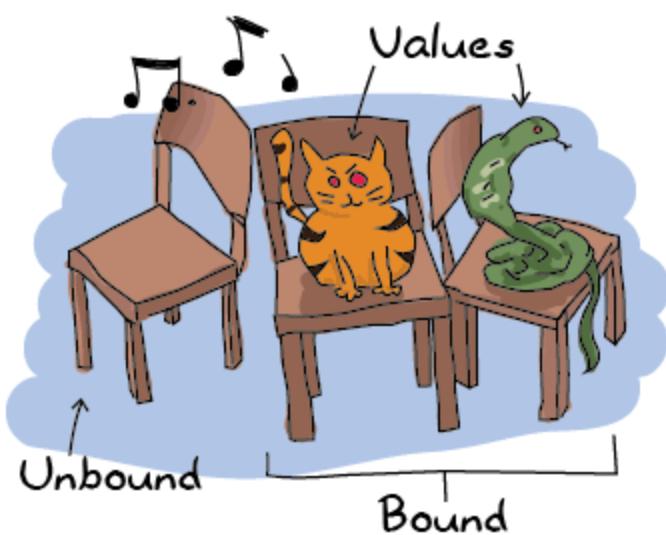
```
1> c(functions).  
{ok, functions}  
2> functions:head([1,2,3,4]).  
1  
3> functions:second([1,2,3,4]).  
2
```

This could be repeated for lists as long as you want, although it would be impractical to do it up to thousands of values.

This can be fixed by writing recursive functions, which we'll see how to do later on. For now, let's concentrate on more pattern matching. The concept of free and bound variables we discussed in [Starting Out \(for real\)](#) still holds true for functions: we can then compare and know if two parameters passed to a function are the same or not. For this, we'll create a function `same/2` that takes two arguments and tells if they're identical:

```
same(X,X) ->  
    true;  
same(_,_) ->  
    false.
```

And it's that simple. Before explaining how the function works, we'll go over the concept of bound and unbound variables again, just in case:



If this game of musical chairs was Erlang, you'd want to sit on the empty chair. Sitting on one already occupied wouldn't end well! Joking aside, unbound variables are variables without any values attached to them (like our empty chair). Binding a variable is simply attaching a value to an unbound variable. In the case of Erlang, when you want to assign a value to a variable that is already bound, an error occurs *unless the new value is the same as the old one*. Let's imagine our snake on the right: if another snake comes around, it won't really change much to the game. You'll just have more angry snakes. If a different animal comes to sit on the chair (a honey badger, for example), things will go bad. Same values for a bound variable are fine, different ones are a bad idea. You can go back to the subchapter about [Invariable Variables](#) if this concept is not clear to you.

Back to our code: what happens when you call `same(a,a)` is that the first `X` is seen as unbound: it automatically takes the value `a`. Then when Erlang goes over to the second argument, it sees `X` is already bound. It then compares it to the `a` passed as the second argument and looks to see if it matches. The pattern matching succeeds and the function returns true. If the two values aren't the same, this will fail and go to the second function clause, which doesn't care about its arguments (when you're the last to choose, you can't be picky!) and will instead return false. Note that this function can effectively take any kind of argument whatsoever! It works for any type of data, not just lists or single variables. As

a rather advanced example, the following function prints a date, but only if it is formatted correctly:

```
valid_time({Date = {Y,M,D}, Time = {H,Min,S}}) ->
    io:format("The Date tuple (~p) says today is: ~p/~p/~p,~n",[Date,Y,M,D]),
    io:format("The time tuple (~p) indicates: ~p:~p:~p.~n", [Time,H,Min,S]);
valid_time(_) ->
    io:format("Stop feeding me wrong data!~n").
```

Note that it is possible to use the = operator in the function head, allowing us to match both the content inside a tuple ({Y,M,D}) and the tuple as a whole (*Date*). The function can be tested the following way:

```
4> c(functions).
{ok, functions}
5> functions:valid_time({{2011,09,06},{09,04,43}}).
The Date tuple ({2011,9,6}) says today is: 2011/9/6,
The time tuple ({9,4,43}) indicates: 9:4:43.
ok
6> functions:valid_time({{2011,09,06},{09,04}}).
Stop feeding me wrong data!
ok
```

There is a problem though! This function could take anything for values, even text or atoms, as long as the tuples are of the form {{A,B,C}, {D,E,F}}. This denotes one of the limits of pattern matching: it can either specify really precise values such as a known number of atom, or abstract values such as the head|tail of a list, a tuple of *N* elements, or anything (_ and unbound variables), etc. To solve this problem, we use guards.

Guards, Guards!



Guards are additional clauses that can go in a function's head to make pattern matching more expressive. As mentioned above, pattern matching is somewhat limited as it cannot express things like a range of value or certain types of data. A concept we couldn't represent is counting: is this 12 years old basketball player too short to play with the pros? Is this distance too long to walk on your hands? Are you too old or too young to drive a car? You couldn't answer these with simple pattern matching. I mean, you could represent the driving question such as:

```
old_enough(0) -> false;  
old_enough(1) -> false;  
old_enough(2) -> false;  
...  
old_enough(14) -> false;  
old_enough(15) -> false;  
old_enough(_) -> true.
```

But it would be incredibly impractical. You can do it if you want, but you'll be alone to work on your code forever. If you want to eventually make friends, start a new guards module

so we can type in the "correct" solution to the driving question:

```
old_enough(X) when X >= 16 -> true;  
old_enough(_) -> false.
```

And you're done! As you can see, this is much shorter and cleaner. Note that a basic rule for guard expression is they must return true to succeed. The guard will fail if it returns false or if it throws an exception. Suppose we now forbid people who are over 104 years old to drive. Our valid ages for drivers is now from 16 years old up to 104 years old. We need to take care of that, but how? Let's just add a second guard clause:

```
right_age(X) when X >= 16, X =< 104 ->  
    true;  
right_age(_) ->  
    false.
```

The comma (,) acts in a similar manner to the operator `andalso` and the semicolon (;) acts a bit like `orelse` (described in "[Starting Out \(for real\)](#)"). Both guard expressions need to succeed for the whole guard to pass. We could also represent the function the opposite way:

```
wrong_age(X) when X < 16; X > 104 ->  
    true;  
wrong_age(_) ->  
    false.
```



And we get correct results from that too. Test it if you want (you should always test stuff!). In guard expressions, the semi-colon (`;`) acts like the `orelse` operator: if the first guard fails, it then tries the second, and then the next one, until either one guard succeeds or they all fail.

You can use a few more functions than comparisons and boolean evaluation in functions, including math operations (`A*B/C >= 0`) and functions about data types, such as `is_integer/1`, `is_atom/1`, etc. (We'll get back on them in the following chapter). One negative point about guards is that they will not accept user-defined functions because of side effects. Erlang is not a purely functional programming language (like Haskell is) because it relies on side effects a lot: you can do I/O, send messages between actors or throw errors as you want and when you want. There is no trivial way to determine if a function you would use in a guard would or wouldn't print text or catch important errors every time it is tested over many function clauses. So instead, Erlang just doesn't trust you (and it may be right to do so!)

That being said, you should be good enough to understand the basic syntax of guards to understand them when you encounter them.

Note: I've compared , and ; in guards to the operators `andalso` and `orelse`. They're not exactly the same, though. The former pair will catch exceptions as they happen while the latter won't. What this means is that if there is an error thrown in the first part of the guard `x >= N; N >= 0`, the second part can still be evaluated and the guard might succeed; if an error was thrown in the first part of `x >= N orelse N >= 0`, the second part will also be skipped and the whole guard will fail.

However (there is always a 'however'), only `andalso` and `orelse` can be nested inside guards. This means `(A orelse B) andalso C` is a valid guard, while `(A; B), C` is not. Given their different use, the best strategy is often to mix them as necessary.

What the If!?

Ifs act like guards and share guards' syntax, but outside of a function clause's head. In fact, the if clauses are called *Guard Patterns*. Erlang's ifs are different from the ifs you'll ever encounter in most other languages; compared to them they're weird creatures that might have been more accepted had they had a different name. When entering Erlang's country, you should leave all you know about ifs at the door. Take a seat because we're going for a ride.

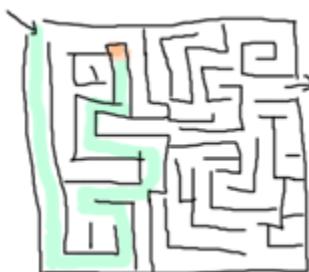
To see how similar to guards the if expression is, look at the following examples:

```
-module(what_the_if).  
-export([heh_fine/0]).
```

```
heh_fine() ->  
    if 1 =:= 1 ->  
        works  
    end,  
    if 1 =:= 2; 1 =:= 1 ->  
        works  
    end,  
    if 1 =:= 2, 1 =:= 1 ->  
        fails  
    end.
```

Save this as what_the_if.erl and let's try it:

```
1> c(what_the_if).  
../what_the_if.erl:12: Warning: no clause will ever match  
../what_the_if.erl:12: Warning: the guard for this clause evaluates to 'false'  
{ok,what_the_if}  
2> what_the_if:heh_fine().  
** exception error: no true branch found when evaluating an if expression  
in function what_the_if:heh_fine/0
```



Uh oh! the compiler is warning us that no clause from the if on line 12 ($1 =:= 2, 1 =:= 1$) will ever match because its only guard evaluates to false. Remember, in Erlang, everything has to return something, and if expressions are no exception to the rule. As such, when Erlang can't find a way to have a guard succeed, it will crash: it cannot *not* return something. As such, we need to add a catch-all branch that will always succeed no matter what. In most languages, this would be called an 'else'. In Erlang, we use 'true' (this explains why the VM has thrown "no true branch found" when it got mad):

```
oh_god(N) ->
    if N =:= 2 -> might_succeed;
        true -> always_does %% this is Erlang's if's 'else!'
    end.
```

And now if we test this new function (the old one will keep spitting warnings, ignore them or take them as a reminder of what not to do):

```
3> c(what_the_if).
./what_the_if.erl:12: Warning: no clause will ever match
./what_the_if.erl:12: Warning: the guard for this clause evaluates to 'false'
{ok,what_the_if}
4> what_the_if:oh_god(2).
might_succeed
5> what_the_if:oh_god(3).
always_does
```

Here's another function showing how to use many guards in an if expression. The function also illustrates how any expression must return something: *Talk* has the result of the if

expression bound to it, and is then concatenated in a string, inside a tuple. When reading the code, it's easy to see how the lack of a true branch would mess things up, considering Erlang has no such thing as a null value (ie.: Lisp's NIL, C's NULL, Python's None, etc):

```
%% note, this one would be better as a pattern match in function heads!
%% I'm doing it this way for the sake of the example.
help_me(Animal) ->
    Talk = if Animal == cat -> "meow";
            Animal == beef -> "mooo";
            Animal == dog -> "bark";
            Animal == tree -> "bark";
            true -> "fgdadfgna"
    end,
{Animal, "says " ++ Talk ++ "!"}.
```

And now we try it:

```
6> c(what_the_if).
./what_the_if.erl:12: Warning: no clause will ever match
./what_the_if.erl:12: Warning: the guard for this clause evaluates to 'false'
{ok,what_the_if}
7> what_the_if:help_me(dog).
{dog,"says bark!"}
8> what_the_if:help_me("it hurts!").
{"it hurts!","says fgdadfgna!"}
```

You might be one of the many Erlang programmers wondering why 'true' was taken over 'else' as an atom to control flow; after all, it's much more familiar. Richard O'Keefe gave the following answer on the Erlang mailing lists. I'm quoting it directly because I couldn't have put it better:

It may be more FAMILIAR, but that doesn't mean 'else' is a good thing. I know that writing '`; true ->`' is a very easy way to get 'else' in Erlang, but we have a couple of decades of psychology-of-programming results to show that it's a bad idea. I have started to replace:

```
by
if X > Y -> a()           if X > Y -> a()
; true -> b()             ; X =< Y -> b()
end           end

if X > Y -> a()           if X > Y -> a()
; X < Y -> b()           ; X < Y -> b()
; true -> c()           ; X == Y -> c()
end           end
```

which I find mildly annoying when writing the code but enormously helpful when reading it.

'Else' or 'true' branches should be "avoided" altogether: ifs are usually easier to read when you cover all logical ends rather than relying on a "*catch all*" clause.

As mentioned before, there are only a limited set of functions that can be used in guard expressions (we'll see more of them in [Types \(or lack thereof\)](#)). This is where the real conditional powers of Erlang must be conjured. I present to you: the case expression!

Note: All this horror expressed by the function names in what_the_if.erl is expressed in regards to the if language

construct when seen from the perspective of any other languages' if. In Erlang's context, it turns out to be a perfectly logical construct with a confusing name.

In Case ... of

If the if expression is like a guard, a case ... of expression is like the whole function head: you can have the complex pattern matching you can use with each argument, and you can have guards on top of it!

As you're probably getting pretty familiar with the syntax, we won't need too many examples. For this one, we'll write the append function for sets (a collection of unique values) that we will represent as an unordered list. This is possibly the worst implementation possible in terms of efficiency, but what we want here is the syntax:

```
insert(X,[]) ->
    [X];
insert(X,Set) ->
    case lists:member(X,Set) of
        true -> Set;
        false -> [X|Set]
    end.
```

If we send in an empty set (list) and a term X to be added, it returns us a list containing only X . Otherwise, the function lists:member/2 checks whether an element is part of a list and returns true if it is, false if it is not. In the case we already had

the element X in the set, we do not need to modify the list. Otherwise, we add X as the list's first element.

In this case, the pattern matching was really simple. It can get more complex (you can compare your code with mine):

```
beach(Temperature) ->
  case Temperature of
    {celsius, N} when N >= 20, N =< 45 ->
      'favorable';
    {kelvin, N} when N >= 293, N =< 318 ->
      'scientifically favorable';
    {fahrenheits, N} when N >= 68, N =< 113 ->
      'favorable in the US';
    _ ->
      'avoid beach'
  end.
```

Here, the answer of "is it the right time to go to the beach" is given in 3 different temperature systems: Celsius, Kelvins and Fahrenheit degrees. Pattern matching and guards are combined in order to return an answer satisfying all uses. As pointed out earlier, `case ... of` expressions are pretty much the same thing as a bunch of function heads with guards. In fact we could have written our code the following way:

```
beachf({celsius, N}) when N >= 20, N =< 45 ->
  'favorable';
...
beachf(_) ->
  'avoid beach'.
```

This raises the question: when should we use if, case ... of or functions to do conditional expressions?



Which to use?

Which to use is rather hard to answer. The difference between function calls and case ... of are very minimal: in fact, they are represented the same way at a lower level, and using one or the other effectively has the same cost in terms of performance. One difference between both is when more than one argument needs to be evaluated: function(A,B) -> ... end. can have guards and values to match against A and B, but a case expression would need to be formulated a bit like:

```
case {A,B} of
    Pattern Guards -> ...
end.
```

This form is rarely seen and might surprise the reader a bit. In similar situations, using a function call might be more appropriate. On the other hand the insert/2 function we had

written earlier is arguably cleaner the way it is rather than having an immediate function call to track down on a simple true or false clause.

Then the other question is why would you ever use if, given cases and functions are flexible enough to even encompass if through guards? The rationale behind if is quite simple: it was added to the language as a short way to have guards without needing to write the whole pattern matching part when it wasn't needed.

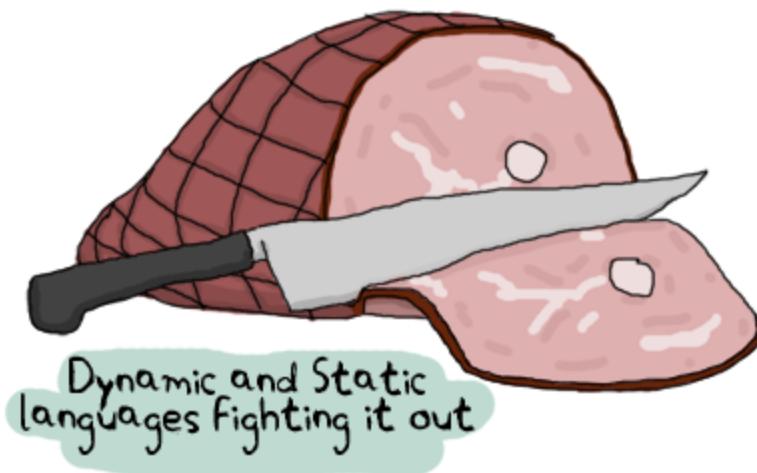
Of course, all of this is more about personal preferences and what you may encounter more often. There is no good solid answer. The whole topic is still debated by the Erlang community from time to time. Nobody's going to go try to beat you up because of what you've chosen, as long as it is easy to understand. As Ward Cunningham once put it, "*Clean code is when you look at a routine and it's pretty much what you expected.*"

Types (or lack thereof)

Dynamite-strong Typing

As you might have noticed when typing in examples from [Starting Out \(for real\)](#), and then modules and functions from [Modules](#) and [Syntax in Functions](#), we never needed to write the type of a variable or the type of a function. When pattern matching, the code we had written didn't have to know what it would be matched against. The tuple {x,y} could be matched with {atom,123} as well as {"A string", <<"binary stuff!">>}, {2.0, ["strings","and",atoms]} or really anything at all.

When it didn't work, an error was thrown in your face, but only once you ran the code. This is because Erlang is *dynamically typed*: every error is caught at runtime and the compiler won't always yell at you when compiling modules where things may result in failure, like in [Starting Out \(for real\)](#)'s "llama + 5" example.



One classic friction point between proponents of static and dynamic typing has to do with the safety of the software being written. A frequently suggested idea is that good static type systems with compilers enforcing them with fervor will catch most errors waiting to happen before you can even execute the code. As such, statically typed languages are to be seen as safer than their dynamic counterparts. While this might be true when comparing with many dynamic languages, Erlang begs to differ and certainly has a track record to prove it. The best example is the often reported *nine nines* (99.999999%) of availability offered on the Ericsson AXD 301 ATM switches, consisting of over 1 million lines of Erlang code. Please note that this is not an indication that none of the components in an Erlang-based system failed, but that a general switch system was available 99.999999% of the time, planned outages included. This is partially because Erlang is built on the notion that a failure in one of the components should not affect the whole system. Errors coming from the programmer, hardware failures or [some] network failures are accounted for: the language includes features which will allow you to distribute a program over to different nodes, handle unexpected errors, and never stop running.

To make it short, while most languages and type systems aim to make a program error-free, Erlang uses a strategy where it is assumed that errors will happen anyway and makes sure to cover these cases: Erlang's dynamic type system is not a barrier to reliability and safety of programs.

This sounds like a lot of prophetic talking, but you'll see how it's done in the later chapters.

Note: Dynamic typing was historically chosen for simple reasons; those who implemented Erlang at first mostly came from dynamically typed languages, and as such, having Erlang dynamic was the most natural option to them.

Erlang is also strongly typed. A weakly typed language would do implicit type conversions between terms. If Erlang were to be weakly typed we could possibly do the operation $6 = 5 + "1"$. while in practice, an exception for bad arguments will be thrown:

```
1> 6 + "1".
```

```
** exception error: bad argument in an arithmetic expression  
in operator +/2  
called as 6 + "1"
```

Of course, there are times when you could want to convert one kind of data to another one: changing regular strings into bit strings to store them or an integer to a floating point number. The Erlang standard library provides a number of functions to do it.

Type conversions

Erlang, like many languages, changes the type of a term by casting it into another one. This is done with the help of built-in functions, as many of the conversions could not be implemented in Erlang itself. Each of these functions take the

form `<type>_to_<type>` and are implemented in the erlang module. Here are a few of them:

```
1> erlang:list_to_integer("54").  
54  
2> erlang:integer_to_list(54).  
"54"  
3> erlang:list_to_integer("54.32").  
** exception error: bad argument  
  in function  list_to_integer/1  
    called as list_to_integer("54.32")  
4> erlang:list_to_float("54.32").  
54.32  
5> erlang:atom_to_list(true).  
"true"  
6> erlang:list_to_bitstring("hi there").  
<<"hi there">>  
7> erlang:bitstring_to_list(<<"hi there">>).  
"hi there"
```

And so on. We're hitting on a language wart here: because the scheme `<type>_to_<type>` is used, every time a new type is added to the language, a whole lot of conversion BIFs need to be added! Here's the whole list already there:

```
atom_to_binary/2, atom_to_list/1, binary_to_atom/2,  
binary_to_existing_atom/2, binary_to_list/1, bitstring_to_list/1,  
binary_to_term/1, float_to_list/1, fun_to_list/1, integer_to_list/1,  
integer_to_list/2, iolist_to_binary/1, iolist_to_atom/1, list_to_atom/1,  
list_to_binary/1, list_to_bitstring/1, list_to_existing_atom/1, list_to_float/1,  
list_to_integer/2, list_to_pid/1, list_to_tuple/1, pid_to_list/1, port_to_list/1,  
ref_to_list/1, term_to_binary/1, term_to_binary/2 and tuple_to_list/1.
```

That's a lot of conversion functions. We'll see most if not all of these types through this book, although we probably won't need all of these functions.

To Guard a Data Type

Erlang basic data types are easy to spot, visually: tuples have the curly brackets, lists the square brackets, strings are enclosed in double quotation marks, etc. Enforcing a certain data type has thus been possible with pattern matching: a function `head/1` taking a list could only accept lists because otherwise, the matching `([H|_])` would have failed.



However, we've had a problem with numeric values because we couldn't specify ranges. Consequently, we used guards in functions about temperature, the age to drive, etc. We're hitting another roadblock now. How could we write a guard that ensures that patterns match against data of a single specific type, like numbers, atoms or bitstrings?

There are functions dedicated to this task. They will take a single argument and return true if the type is right, false otherwise. They are part of the few functions allowed in guard expressions and are named the *type test BIFs*:

`is_atom/1` `is_binary/1`
`is_bitstring/1` `is_boolean/1` `is_builtin/3`

```
is_float/1      is_function/1    is_function/2  
is_integer/1    is_list/1       is_number/1  
is_pid/1        is_port/1      is_record/2  
is_record/3     is_reference/1   is_tuple/1
```

They can be used like any other guard expression, wherever guard expressions are allowed. You might be wondering why there is no function just giving the type of the term being evaluated (something akin to `type_of(X) -> Type`). The answer is pretty simple. Erlang is about programming for the right cases: you only program for what you know will happen and what you expect. Everything else should cause errors as soon as possible. Although this might sound insane, the explanations you'll get in [Errors and Exceptions](#) will hopefully make things clearer. Until then, just trust me on that.

Note: type test BIFs constitute more than half of the functions allowed in guard expressions. The rest are also BIFs, but do not represent type tests. These are:

```
abs(Number), bit_size(Bitstring), byte_size(Bitstring), element(N, Tuple),  
float(Term), hd(List), length(List), node(), node(Pid|Ref|Port),  
round(Number), self(), size(Tuple|Bitstring), tl(List), trunc(Number),  
tuple_size(Tuple).
```

The functions `node/1` and `self/0` are related to distributed Erlang and processes/actors. We'll eventually use them, but we've still got other topics to cover before then.

It may seem like Erlang data structures are relatively limited, but lists and tuples are usually enough to build other

complex structures without worrying about anything. As an example the basic node of a binary tree could be represented as {node, Value, Left, Right}, where *Left* and *Right* are either similar nodes or empty tuples. I could also represent myself as:

```
{person, {name, <<"Fred T-H">>},  
  {qualities, ["handsome", "smart", "honest", "objective"]},  
  {faults, ["liar"]},  
  {skills, ["programming", "bass guitar", "underwater breakdancing"]}}.
```

Which shows that by nesting tuples and lists and filling them with data, we can obtain complex data structures and build functions to operate on them.

Update:

The release R13B04 saw the addition of the BIF `binary_to_term/2`, which lets you unserialize data the same way `binary_to_term/1` would, except the second argument is an option list. If you pass in `[safe]`, the binary won't be decoded if it contains unknown atoms or [anonymous functions](#), which could exhaust memory.

For Type Junkies



This section is meant to be read by programmers who can not live without a static type system for one reason or another. It will include a little bit more advanced theory and everything may not be understood by everyone. I will briefly describe tools used to do static type analysis in Erlang, defining custom types and getting more safety that way. These tools will be described for anyone to understand much later in the book, given that it is not necessary to use any of them to write reliable Erlang programs. Because we'll show them later, I'll give very little details about installing, running them, etc. Again, this section is for those who really can't live without advanced type systems.

Through the years, there were some attempts to build type systems on top of Erlang. One such attempt happened back in 1997, conducted by Simon Marlow, one of the lead developers of the Glasgow Haskell Compiler, and Philip Wadler, who worked on Haskell's design and has contributed to the theory behind monads (Read the paper on said type system). Joe Armstrong later commented on the paper:

One day Phil phoned me up and announced that a) Erlang needed a type system, b) he had written a small prototype of a type system and c) he had a one year's sabbatical and was going to write a type system for Erlang and "were we interested?" Answer —"Yes."

Phil Wadler and Simon Marlow worked on a type system for over a year and the results were published in [20].

The results of the project were somewhat disappointing. To start with, only a subset of the language was type-checkable, the major omission being the lack of process types and of type checking inter-process messages.

Processes and messages both being one of the core features of Erlang, it may explain why the system was never added to the language. Other attempts at typing Erlang failed. The efforts of the HiPE project (attempts to make Erlang's performances much better) produced Dialyzer, a static analysis tool still in use today, with its very own type inference mechanism.

The type system that came out of it is based on success typings, a concept different from Hindley–Milner or soft-typing type systems. Success types are simple in concept: the type-inference will not try to find the exact type of every expression, but it will guarantee that the types it infers are right, and that the type errors it finds are really errors.

The best example would come from the implementation of the function `and`, which will usually take two Boolean values and return 'true' if they're both true, 'false' otherwise. In Haskell's type system, this would be written `and :: bool -> bool -> bool`. If the `and` function had to be implemented in Erlang, it could be done the following way:

```
and(false, _) -> false;  
and(_, false) -> false;  
and(true,true) -> true.
```

Under success typing, the inferred type of the function would be `and(_,_)` → `bool()`, where `_` means 'anything'. The reason for this is simple: when running an Erlang program and calling this function with the arguments `false` and `42`, the result would still be 'false'. The use of the `_` wildcard in pattern matching made it that in practice, any argument can be passed as long as one of them is 'false' for the function to work. ML types would have thrown a fit (and its users had a heart attack) if you had called the function this way. Not Erlang. It might make more sense to you if you decide to read the paper on the implementation of success types, which explains the rationale behind the behavior. I really encourage any type junkies out there to read it, it's an interesting and practical implementation definition.

The details about type definitions and function annotations are described in the Erlang Enhancement Proposal 8 (EEP 8). If you're interested in using success typings in Erlang, check out the TypEr application and Dialyzer, both part of the standard distribution. To use them, type in `$ typer --help` and `$ dialyzer --help` (`typer.exe --help` and `dialyzer.exe --help` for Windows, if they're accessible from the directory you are currently in).

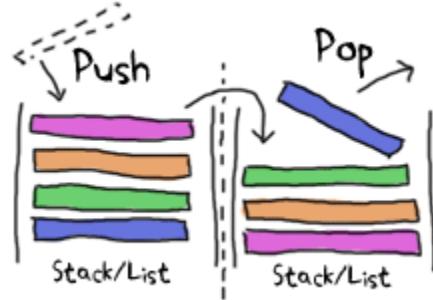
TypEr will be used to generate type annotations for functions. Used on this small FIFO implementation, it spits the following type annotations:

```
%% File: fifo.erl
%% -----
-spec new() -> {'fifo',[],[]}.
```

```
-spec push({'fifo',_,_},_) -> {'fifo',nonempty_maybe_improper_list(),_}.
```

```
-spec pop({'fifo',_,maybe_improper_list()}) -> {_,{'fifo',_,_}}.
```

```
-spec empty({'fifo',_,_}) -> bool().
```



Which is pretty much right. Improper lists should be avoided because `lists:reverse/1` doesn't support them, but someone bypassing the module's interface would be able to get through it and submit one. In this case, the functions `push/2` and `pop/2` might still succeed for a few calls before they cause an exception. This either tells us to add guards or refine our type definitions manually. Suppose we add the signature `-spec push({'fifo,list(),list()},_) -> {'fifo,nonempty_list(),list()}.` and a function that passes an improper list to `push/2` to the module: when scanning it in Dialyzer (which checks and matches the types), the error message "The call `fifo:push({fifo, [1|2],[]},3)` breaks the contract '`<Type definition here>`' is output.

Dialyzer will complain only when code will break other code, and if it does, it'll usually be right (it will complain about more stuff too, like clauses that will never match or general discrepancies). Polymorphic data types are also possible to write and analyze with Dialyzer: the `hd()` function could be annotated with `-spec([A]) -> A.` and be analyzed correctly,

although Erlang programmers seem to rarely use this type syntax.

Don't drink too much Kool-Aid:

Some of the things you can't expect Dialyzer and TypEr to do is type classes with constructors, first order types and recursive types. The types of Erlang are only annotations without effects or restrictions on actual compiling unless you enforce them yourself. The type checker will never tell you a program that can run right now (or has run for two years) has a type bug when it effectively causes no error when running (although you could have buggy code running correctly...)

While recursive types are something that would be really interesting to have, they're unlikely to ever appear in the current forms of TypEr and Dialyzer (the paper above explains why). Defining your own types to simulate recursive types by adding one or two levels manually is the best you can do at the moment.

It's certainly not a full-blown type system, not as strict or powerful as what languages like Scala, Haskell or Ocaml propose. Its warning and error messages are also usually a bit cryptic and not really user friendly. However, it's still a very good compromise if you really can't live in a dynamic world or wish for additional safety; just expect it to be a tool in your arsenal, not too much more.

Update:

Since version R13B04, recursive types are now available as an experimental feature for Dialyzer. This makes the previous *Don't drink too much Kool-aid* partially wrong. Shame on me.

Note that the type documentation has also become official (although it remains subject to change) and is more complete than what can be found in EEP8.

Recursion

Hello recursion!



Some readers accustomed with imperative and object-oriented programming languages might be wondering why loops weren't shown already. The answer to this is "what is a loop?" Truth is, functional programming languages usually do not offer looping constructs like `for` and `while`. Instead, functional programmers rely on a silly concept named *recursion*.

I suppose you remember how invariable variables were explained in the intro chapter. If you don't, you can [give them more attention](#)! Recursion can also be explained with the help of mathematical concepts and functions. A basic mathematical function such as the factorial of a value is a good example of a function that can be expressed recursively. The factorial of a number n is the product of the sequence $1 \times 2 \times 3 \times \dots \times n$, or alternatively $n \times (n-1) \times (n-2) \times \dots \times 1$. To give some examples, the factorial of 3 is $3! = 3 \times 2 \times 1 = 6$. The factorial of 4 would be $4! = 4 \times 3 \times 2 \times 1 = 24$. Such a function can be expressed the following way in mathematical notation:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n((n-1)!) & \text{if } n > 0 \end{cases}$$

What this tells us is that if the value of n we have is 0, we return the result 1. For any value above 0, we return n multiplied by the factorial of $n-1$, which unfolds until it reaches 1:

$$\begin{aligned} 4! &= 4 \times 3! \\ 4! &= 4 \times 3 \times 2! \\ 4! &= 4 \times 3 \times 2 \times 1! \\ 4! &= 4 \times 3 \times 2 \times 1 \times 1 \end{aligned}$$

How can such a function be translated from mathematical notation to Erlang? The conversion is simple enough. Take a look at the parts of the notation: $n!$, 1 and $n((n-1)!)$ and then the ifs. What we've got here is a function name ($n!$), guards (the ifs) and a the function body (1 and $n((n-1)!)$). We'll rename $n!$ to `fac(N)` to restrict our syntax a bit and then we get the following:

```
-module(recursive).  
-export([fac/1]).
```

```
fac(N) when N == 0 -> 1;  
fac(N) when N > 0 -> N*fac(N-1).
```

And this factorial function is now done! It's pretty similar to the mathematical definition, really. With the help of pattern matching, we can shorten the definition a bit:

```
fac(0) -> 1;  
fac(N) when N > 0 -> N*fac(N-1).
```

So that's quick and easy for some mathematical definitions which are recursive in nature. We looped! A definition of recursion could be made short by saying "a function that calls itself." However, we need to have a stopping condition (the real term is *base case*), because we'd otherwise loop infinitely. In our case, the stopping condition is when *n* is equal to 0. At that point we no longer tell our function to call itself and it stops its execution right there.

Length

Let's try to make it slightly more practical. We'll implement a function to count how many elements a list contains. So we know from the beginning that we will need:

- a base case;
- a function that calls itself;
- a list to try our function on.

With most recursive functions, I find the base case easier to write first: what's the simplest input we can have to find a length from? Surely an empty list is the simplest one, with a length of 0. So let's make a mental note that $[] = 0$ when dealing with lengths. Then the next simplest list has a length of 1: $[_] = 1$. This sounds like enough to get going with our definition. We can write this down:

```
len([]) -> 0;  
len([_]) -> 1.
```

Awesome! We can calculate the length of lists, given the length is either 0 or 1! Very useful indeed. Well of course it's useless, because it's not yet recursive, which brings us to the hardest part: extending our function so it calls itself for lists longer than 1 or 0. It was [mentioned earlier](#) that lists are defined recursively as $[1 | [2 | \dots | n | []]]$. This means we can use the $[H|T]$ pattern to match against lists of one or more elements, as a list of length one will be defined as $[x|[]]$ and a list of length two will be defined as $[x|[y|[]]]$. Note that the second element is a list itself. This means we only need to count the first one and the function can call itself on the second element. Given each value in a list counts as a length of 1, the function can be rewritten the following way:

```
len([]) -> 0;  
len([_|T]) -> 1 + len(T).
```

And now you've got your own recursive function to calculate the length of a list. To see how `len/1` would behave when ran, let's try it on a given list, say `[1,2,3,4]`:

```

len([1,2,3,4]) = len([1 | [2,3,4]])
= 1 + len([2 | [3,4]])
= 1 + 1 + len([3 | [4]])
= 1 + 1 + 1 + len([4 | []])
= 1 + 1 + 1 + 1 + len([])
= 1 + 1 + 1 + 1 + 0
= 1 + 1 + 1 + 1
= 1 + 1 + 2
= 1 + 3
= 4

```

Which is the right answer. Congratulations on your first useful recursive function in Erlang!



Length of a Tail Recursion

You might have noticed that for a list of 4 terms, we expanded our function call to a single chain of 5 additions. While this does the job fine for short lists, it can become problematic if your list has a few million values in it. You don't want to keep millions of numbers in memory for such a simple calculation. It's wasteful and there's a better way. Enter *tail recursion*.

Tail recursion is a way to transform the above linear process (it grows as much as there are elements) to an iterative one (there is not really any growth). To have a function call being tail recursive, it needs to be 'alone'. Let me explain: what made our previous calls grow is how the answer of the first part depended on evaluating the second part. The answer to $1 + \text{len}(\text{Rest})$ needs the answer of $\text{len}(\text{Rest})$ to be found. The function $\text{len}(\text{Rest})$ itself then needed the result of another function call to be found. The additions would get stacked until the last one is found, and only then would the final result be calculated. Tail recursion aims to eliminate this stacking of operation by reducing them as they happen.

In order to achieve this, we will need to hold an extra temporary variable as a parameter in our function. I'll illustrate the concept with the help of the factorial function, but this time defining it to be tail recursive. The aforementioned temporary variable is sometimes called *accumulator* and acts as a place to store the results of our computations as they happen in order to limit the growth of our calls:

```
tail_fac(N) -> tail_fac(N,1).
```

```

tail_fac(0,Acc) -> Acc;
tail_fac(N,Acc) when N > 0 -> tail_fac(N-1,N*Acc).

```

Here, I define both `tail_fac/1` and `tail_fac/2`. The reason for this is that Erlang doesn't allow default arguments in functions (different arity means different function) so we do that

manually. In this specific case, `tail_fac/1` acts like an abstraction over the tail recursive `tail_fac/2` function. The details about the hidden accumulator of `tail_fac/2` don't interest anyone, so we would only export `tail_fac/1` from our module. When running this function, we can expand it to:

```
tail_fac(4) = tail_fac(4,1)
tail_fac(4,1) = tail_fac(4-1, 4*1)
tail_fac(3,4) = tail_fac(3-1, 3*4)
tail_fac(2,12) = tail_fac(2-1, 2*12)
tail_fac(1,24) = tail_fac(1-1, 1*24)
tail_fac(0,24) = 24
```

See the difference? Now we never need to hold more than two terms in memory: the space usage is constant. It will take as much space to calculate the factorial of 4 as it will take space to calculate the factorial of 1 million (if we forget $4!$ is a smaller number than $1M!$ in its complete representation, that is).

With an example of tail recursive factorials under your belt, you might be able to see how this pattern could be applied to our `len/1` function. What we need is to make our recursive call 'alone'. If you like visual examples, just imagine you're going to put the `+1` part inside the function call by adding a parameter:

```
len([]) -> 0;
len([_ | T]) -> 1 + len(T).
```

becomes:

```
tail_len(L) -> tail_len(L,0).
```

```
tail_len([], Acc) -> Acc;
tail_len([_ | T], Acc) -> tail_len(T, Acc+1).
```

And now your length function is tail recursive.

More recursive functions



We'll write a few more recursive functions, just to get in the habit a bit more. After all, recursion being the only looping construct that exists in Erlang (except list comprehensions),

it's one of the most important concepts to understand. It's also useful in every other functional programming language you'll try afterwards, so take notes!

The first function we'll write will be `duplicate/2`. This function takes an integer as its first parameter and then any other term as its second parameter. It will then create a list of as many copies of the term as specified by the integer. Like before, thinking of the base case first is what might help you get going. For `duplicate/2`, asking to repeat something 0 time is the most basic thing that can be done. All we have to do is return an empty list, no matter what the term is. Every other case needs to try and get to the base case by calling the function itself. We will also forbid negative values for the integer, because you can't duplicate something $-n$ times:

```
duplicate(0,_) ->
[];;
duplicate(N,Term) when N > 0 ->
[Term|duplicate(N-1,Term)].
```

Once the basic recursive function is found, it becomes easier to transform it into a tail recursive one by moving the list construction into a temporary variable:

```
tail_duplicate(N,Term) ->
tail_duplicate(N,Term,[]).

tail_duplicate(0,_,List) ->
List;
tail_duplicate(N,Term,List) when N > 0 ->
tail_duplicate(N-1, Term, [Term|List]).
```

Success! I want to change the subject a little bit here by drawing a parallel between tail recursion and a while loop. Our `tail_duplicate/2` function has all the usual parts of a while loop. If we were to imagine a while loop in a fictional language with Erlang-like syntax, our function could look a bit like this:

```
function(N, Term) ->
while N > 0 ->
List = [Term|List],
N = N-1
end,
List.
```

Note that all the elements are there in both the fictional language and in Erlang. Only their position changes. This demonstrates that a proper tail recursive function is similar to an iterative process, like a while loop.

There's also an interesting property that we can 'discover' when we compare recursive and tail recursive functions by writing a `reverse/1` function, which will reverse a list of terms. For such a function, the base case is an empty list, for which we have nothing to reverse. We can just return an empty list when that happens. Every other possibility should try to

converge to the base case by calling itself, like with `duplicate/2`. Our function is going to iterate through the list by pattern matching `[H|T]` and then putting `H` after the rest of the list:

```
reverse([]) -> [];
reverse([H|T]) -> reverse(T)++[H].
```

On long lists, this will be a true nightmare: not only will we stack up all our append operations, but we will need to traverse the whole list for every single of these appends until the last one! For visual readers, the many checks can be represented as:

```
reverse([1,2,3,4]) = [4]++[3]++[2]++[1]
                     ^
                     = [4,3]++[2]++[1]
                     ^
                     =
                     = [4,3,2]++[1]
                     ^
                     =
                     = [4,3,2,1]
```

This is where tail recursion comes to the rescue. Because we will use an accumulator and will add a new head to it every time, our list will automatically be reversed. Let's first see the implementation:

```
tail_reverse(L) -> tail_reverse(L,[]).
```

```
tail_reverse([],Acc) -> Acc;
tail_reverse([H|T],Acc) -> tail_reverse(T, [H|Acc]).
```

If we represent this one in a similar manner as the normal version, we get:

```
tail_reverse([1,2,3,4]) = tail_reverse([2,3,4], [1])
                     = tail_reverse([3,4], [2,1])
                     = tail_reverse([4], [3,2,1])
                     = tail_reverse([], [4,3,2,1])
                     = [4,3,2,1]
```

Which shows that the number of elements visited to reverse our list is now linear: not only do we avoid growing the stack, we also do our operations in a much more efficient manner!

Another function to implement could be `sublist/2`, which takes a list `L` and an integer `N`, and returns the `N` first elements of the list. As an example, `sublist([1,2,3,4,5,6],3)` would return `[1,2,3]`. Again, the base case is trying to obtain 0 elements from a list. Take care however, because `sublist/2` is a bit different. You've got a second base case when the list passed is empty! If we do not check for empty lists, an error would be thrown when calling `recursive:sublist([1],2)`. while we want `[1]` instead. Once this is defined, the recursive part of the function only has to cycle through the list, keeping elements as it goes, until it hits one of the base cases:

```
sublist(_,0) -> [];
sublist([],_) -> [];
sublist([H|T],N) when N > 0 -> [H|sublist(T,N-1)].
```

Which can then be transformed to a tail recursive form in the same manner as before:

```
tail_sublist(L, N) -> tail_sublist(L, N, []).
```

```
tail_sublist(_, 0, SubList) -> SubList;
tail_sublist([], _, SubList) -> SubList;
tail_sublist([H|T], N, SubList) when N > 0 ->
    tail_sublist(T, N-1, [H|SubList]).
```

There's a flaw in this function. A *fatal flaw!* We use a list as an accumulator in exactly the same manner we did to reverse our list. If you compile this function as is, `sublist([1,2,3,4,5,6],3)` would not return `[1,2,3]`, but `[3,2,1]`. The only thing we can do is take the final result and reverse it ourselves. Just change the `tail_sublist/2` call and leave all our recursive logic intact:

```
tail_sublist(L, N) -> reverse(tail_sublist(L, N, [])).
```

The final result will be ordered correctly. It might seem like reversing our list after a tail recursive call is a waste of time and you would be partially right (we still save memory doing this). On shorter lists, you might find your code is running faster with normal recursive calls than with tail recursive calls for this reason, but as your data sets grow, reversing the list will be comparatively lighter.

Note: instead of writing your own `reverse/1` function, you should use `lists:reverse/1`. It's been used so much for tail recursive calls that the maintainers and developers of Erlang decided to turn it into a BIF. Your lists can now benefit from extremely fast reversal (thanks to functions written in C) which will make the reversal disadvantage a lot less obvious. The rest of the code in this chapter will make use of our own reversal function, but after that you should not use it ever again.

To push things a bit further, we'll write a zipping function. A zipping function will take two lists of same length as parameters and will join them as a list of tuples which all hold two terms. Our own `zip/2` function will behave this way:

```
1> recursive:zip([a,b,c],[1,2,3]).  
[{a,1},{b,2},{c,3}]
```

Given we want our parameters to both have the same length, the base case will be zipping two empty lists:

```
zip([],[]) -> [];  
zip([X|Xs],[Y|Ys]) -> [{X,Y}|zip(Xs,Ys)].
```

However, if you wanted a more lenient zip function, you could decide to have it finish whenever one of the two list is done. In this scenario, you therefore have two base cases:

```
lenient_zip([],_) -> [];  
lenient_zip(_,[ ]) -> [];  
lenient_zip([X|Xs],[Y|Ys]) -> [{X,Y}|lenient_zip(Xs,Ys)].
```

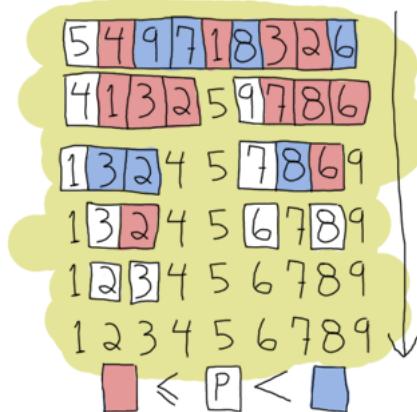
Notice that no matter what our base cases are, the recursive part of the function remains the same. I would suggest you try and make your own tail recursive versions of `zip/2` and `lenient_zip/2`, just to make sure you fully understand how to make tail recursive functions: they'll be one of the central concepts of larger applications where our main loops will be made that way.

If you want to check your answers, take a look at my implementation of `recursive.erl`, more precisely the `tail_zip/2` and `tail_lenient_zip/3` functions.

Note: tail recursion as seen here is not making the memory grow because when the virtual machine sees a function calling itself in a tail position (the last expression to be evaluated in a function), it eliminates the current stack frame. This is called tail-call optimisation (TCO) and it is a special case of a more general optimisation named *Last Call Optimisation* (LCO).

LCO is done whenever the last expression to be evaluated in a function body is another function call. When that happens, as with TCO, the Erlang VM avoids storing the stack frame. As such tail recursion is also possible between multiple functions. As an example, the chain of functions `a() -> b(). b() -> c(). c() -> a()`. will effectively create an infinite loop that won't go out of memory as LCO avoids overflowing the stack. This principle, combined with our use of accumulators is what makes tail recursion useful.

Quick, Sort!



I can (and will) now assume recursion and tail recursion make sense to you, but just to make sure, I'm going to push for a more complex example, quicksort. Yes, the traditional "hey look I can write short functional code" canonical example. A naive implementation of quicksort works by taking the first element of a list, the *pivot*, and then putting all the elements smaller or equal to the pivot in a new list, and all those larger in another list. We then take each of these lists and do the same thing on them until each list gets smaller and smaller. This goes on until you have nothing but an empty list to sort, which will be our base case. This implementation is said to be naive because smarter versions of quicksort will try to pick optimal pivots to be faster. We don't really care about that for our example though.

We will need two functions for this one: a first function to partition the list into smaller and larger parts and a second function to apply the partition function on each of the new lists and to glue them together. First of all, we'll write the glue function:

```
quicksort([]) -> [];
quicksort([Pivot|Rest]) ->
    {Smaller, Larger} = partition(Pivot, Rest, [], []),
    quicksort(Smaller) ++ [Pivot] ++ quicksort(Larger).
```

This shows the base case, a list already partitioned in larger and smaller parts by another function, the use of a pivot with both lists quicksorted appended before and after it. So this should take care of assembling lists. Now the partitioning function:

```
partition(_,[], Smaller, Larger) -> {Smaller, Larger};
partition(Pivot, [H|T], Smaller, Larger) ->
    if H <= Pivot -> partition(Pivot, T, [H|Smaller], Larger);
    H > Pivot -> partition(Pivot, T, Smaller, [H|Larger])
end.
```

And you can now run your quicksort function. If you've looked for Erlang examples on the Internet before, you might have seen another implementation of quicksort, one that is simpler and easier to read, but makes use of list comprehensions. The easy to replace parts are the ones that create new lists, the partition/4 function:

```
lc_quicksort([]) -> [];
lc_quicksort([Pivot|Rest]) ->
    lc_quicksort([Smaller || Smaller <- Rest, Smaller <= Pivot])
    ++
    [Pivot]
    ++
    lc_quicksort([Larger || Larger <- Rest, Larger > Pivot]).
```

The main differences are that this version is much easier to read, but in exchange, it has to traverse the list twice to partition it in two parts. This is a fight of clarity against performance, but the real loser here is you, because a function `lists:sort/1` already exists. Use that one instead.

Don't drink too much Kool-Aid:

All this conciseness is good for educational purposes, but not for performance. Many functional programming tutorials never mention this! First of all, both implementations here need to process values that are equal to the pivot more than once. We could have decided to instead return 3 lists: elements smaller, larger and equal to the pivot in order to make this more efficient.

Another problem relates to how we need to traverse all the partitioned lists more than once when attaching them to the pivot. It is possible to reduce the overhead a little by doing the concatenation while partitioning the lists in three parts. If you're curious about this, look at the last function (`bestest_qsrt/1`) of `recursive.erl` for an example.

A nice point about all of these quicksorts is that they will work on lists of any data type you've got, even tuples of lists and whatnot. Try them, they work!

More than lists

By reading this chapter, you might be starting to think recursion in Erlang is mainly a thing concerning lists. While lists are a good example of a data structure that can be defined recursively, there's certainly more than that. For the sake of diversity, we'll see how to build binary trees, and then read data from them.



First of all, it's important to define what a tree is. In our case, it's nodes all the way down. Nodes are tuples that contain a key, a value associated to the key, and then two other nodes. Of these two nodes, we need one that has a smaller and one that has a larger key than the node holding them. So here's recursion! A tree is a node containing nodes, each of which contains nodes, which in turn also contain nodes. This can't keep going forever (we don't have infinite data to store), so we'll say that our nodes can also contain empty nodes.

To represent nodes, tuples are an appropriate data structure. For our implementation, we can then define these tuples as `{node, {Key, Value, Smaller, Larger}}` (a tagged tuple!), where `Smaller` and `Larger` can be another similar node or an empty node (`{node, nil}`). We won't actually need a concept more complex than that.

Let's start building a module for our very basic tree implementation. The first function, `empty/0`, returns an empty node. The empty node is the starting point of a new tree, also called the *root*:

```
-module(tree).  
-export([empty/0, insert/3, lookup/2]).  
  
empty() -> {node, 'nil'}.
```

By using that function and then encapsulating all representations of nodes the same way, we hide the implementation of the tree so people don't need to know how it's built. All that information can be contained by the module alone. If you ever decide to change the representation of a node, you can then do it without breaking external code.

To add content to a tree, we must first understand how to recursively navigate through it. Let's proceed in the same way as we did for every other recursion example by trying to find the base case. Given that an empty tree is an empty node, our base case is thus logically an empty node. So whenever we'll hit an empty node, that's where we can add our new key/value. The rest of the time, our code has to go through the tree trying to find an empty node where to put content.

To find an empty node starting from the root, we must use the fact that the presence of *Smaller* and *Larger* nodes let us navigate by comparing the new key we have to insert to the current node's key. If the new key is smaller than the current node's key, we try to find the empty node inside *Smaller*, and if it's larger, inside *Larger*. There is one last case, though: what if the new key is equal to the current node's key? We have two options there: let the program fail or replace the value with the new one. This is the option we'll take here. Put into a function all this logic works the following way:

```
insert(Key, Val, {node, 'nil'}) ->
    {node, {Key, Val, {node, 'nil'}, {node, 'nil'}}};

insert(NewKey, NewVal, {node, {Key, Val, Smaller, Larger}}) when NewKey < Key ->
    {node, {Key, Val, insert(NewKey, NewVal, Smaller), Larger}};

insert(NewKey, NewVal, {node, {Key, Val, Smaller, Larger}}) when NewKey > Key ->
    {node, {Key, Val, Smaller, insert(NewKey, NewVal, Larger)}};

insert(Key, Val, {node, {Key, _, Smaller, Larger}}) ->
    {node, {Key, Val, Smaller, Larger}}.
```

Note here that the function returns a completely new tree. This is typical of functional languages having only single assignment. While this can be seen as inefficient, most of the underlying structures of two versions of a tree sometimes happen to be the same and are thus shared, copied by the VM only when needed.

What's left to do on this example tree implementation is creating a `lookup/2` function that will let you find a value from a tree by giving its key. The logic needed is extremely similar to the one used to add new content to the tree: we step through the nodes, checking if the lookup key is equal, smaller or larger than the current node's key. We have two base cases: one when the node is empty (the key isn't in the tree) and one when the key is found. Because we don't want our program to crash each time we look for a key that doesn't exist, we'll return the atom '`undefined`'. Otherwise, we'll return `{ok, value}`. The reason for this is that if we only returned `Value` and the node contained the atom '`undefined`', we would have no way to know if the tree did return the right value or failed to find it. By wrapping successful cases in such a tuple, we make it easy to understand which is which. Here's the implemented function:

```
lookup(_, {node, 'nil'}) ->
    undefined;

lookup(Key, {node, {Key, Val, _, _}}) ->
    {ok, Val};

lookup(Key, {node, {NodeKey, _, Smaller, _}}) when Key < NodeKey ->
    lookup(Key, Smaller);
```

```
lookup(Key, {node, {_, _, _, Larger}}) ->
    lookup(Key, Larger).
```

And we're done. Let's test it with by making a little email address book. Compile the file and start the shell:

```
1> T1 = tree:insert("Jim Woodland", "jim.woodland@gmail.com", tree:empty()).
{node,{ "Jim Woodland", "jim.woodland@gmail.com",
        {node,nil},
        {node,nil}}}
2> T2 = tree:insert("Mark Anderson", "i.am.a@hotmail.com", T1).
{node,{ "Jim Woodland", "jim.woodland@gmail.com",
        {node,nil},
        {node,{ "Mark Anderson", "i.am.a@hotmail.com",
                {node,nil},
                {node,nil}}}}}
3> Addresses = tree:insert("Anita Bath", "abath@someuni.edu", tree:insert("Kevin Robert", "myfairy@yahoo.com", tree:insert("Wilson Longbrow", "longwil@gmail.com", tree:empty()).
{node,{ "Anita Bath", "abath@someuni.edu",
        {node,nil},
        {node,nil}}},
{node,{ "Mark Anderson", "i.am.a@hotmail.com",
        {node,{ "Kevin Robert", "myfairy@yahoo.com",
                {node,nil},
                {node,nil}}}},
{node,{ "Wilson Longbrow", "longwil@gmail.com",
        {node,nil},
        {node,nil}}}}})}
```

And now you can lookup email addresses with it:

```
4> tree:lookup("Anita Bath", Addresses).
{ok, "abath@someuni.edu"}
5> tree:lookup("Jacques Requin", Addresses).
undefined
```

That concludes our functional address book example built from a recursive data structure other than a list! *Anita Bath* now...

Note: Our tree implementation is very naive: we do not support common operations such as deleting nodes or rebalancing the tree to make the following lookups faster. If you're interested in implementing and/or exploring these, studying the implementation of Erlang's `gb_trees` module (`otp_src_R<version>B<revision>/lib/stdlib/src/gb_trees.erl`) is a good idea. This is also the module you should use when dealing with trees in your code, rather than reinventing your own wheel.

Thinking recursively

If you've understood everything in this chapter, thinking recursively is probably becoming more intuitive. A different aspect of recursive definitions when compared to their imperative

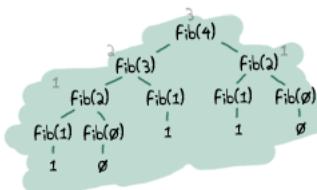
counterparts (usually in while or for loops) is that instead of taking a step-by-step approach ("do this, then that, then this, then you're done"), our approach is more declarative ("if you get this input, do that, this otherwise"). This property is made more obvious with the help of pattern matching in function heads.

If you still haven't grasped how recursion works, maybe reading [this](#) will help you.

Joking aside, recursion coupled with pattern matching is sometimes an optimal solution to the problem of writing concise algorithms that are easy to understand. By subdividing each part of a problem into separate functions until they can no longer be simplified, the algorithm becomes nothing but assembling a bunch of correct answers coming from short routines (that's a bit similar to what we did with quicksort). This kind of mental abstraction is also possible with your everyday loops, but I believe the practice is easier with recursion. Your mileage may vary.

And now ladies and gentlemen, a discussion: the author vs. himself

- — Okay, I think I understand recursion. I get the declarative aspect of it. I get it has mathematical roots, like with invariable variables. I get that you find it easier in some cases. What else?
- — It respects a regular pattern. Find the base cases, write them down, then every other cases should try to converge to these base cases to get your answer. It makes writing functions pretty easy.
- — Yeah, I got that, you repeated it a bunch of times already. My loops can do the same.
- — Yes they can. Can't deny that!
- — Right. A thing I don't get is why you bothered writing all these non-tail recursive versions if they're not as good as tail recursive ones.
- — Oh it's simply to make things easier to grasp. Moving from regular recursion, which is prettier and easier to understand, to tail recursion, which is theoretically more efficient, sounded like a good way to show all options.
- — Right, so they're useless except for educational purposes, I get it.
- — Not exactly. In practice you'll see little difference in the performance between tail recursive and normal recursive calls. The areas to take care of are in functions that are supposed to loop infinitely, like main loops. There's also a type of functions that will always generate very large stacks, be slow and possibly crash early if you don't make them tail recursive. The best example of this is the Fibonacci function, which grows exponentially if it's not iterative or tail recursive.



You should profile your code (I'll show how to do that at a later point, I promise), see what slows it down, and fix it.

- — But loops are always iterative and make this a non-issue.
- — Yes, but... but... my beautiful Erlang..
- — Well isn't that great? All that learning because there is no 'while' or 'for' in Erlang. Thank you very much I'm going back to programming my toaster in C!
- — Not so fast there! Functional programming languages have other assets! If we've found some base case patterns to make our life easier when writing recursive functions, a bunch of smart people have found many more to the point where you will need to write very few recursive functions yourself. If you stay around, I'll show you how such abstractions can be built. But for this we will need more power. Let me tell you about higher order functions...

Higher Order Functions

Let's get functional



An important part of all functional programming languages is the ability to take a function you defined and then pass it as a parameter to another function. This in turn binds that function parameter to a variable which can be used like any other variable within the function. A function that can accept other functions transported around that way is named a *higher order function*. Higher order functions are a powerful means of abstraction and one of the best tools to master in Erlang.

Again, this a concept rooted in mathematics, mainly lambda calculus. I won't go into much detail about lambda calculus because some people have a hard time grasping it and it's a bit out of scope. However, I'll define it briefly as a system where everything is a function, even numbers. Because everything is a function, functions must accept other functions as parameters and can operate on them with even more functions!

Alright, this might be a little bit weird, so let's start with an example:

```
-module(hhfun).
```

```
-compile(export_all).
```

```
one() -> 1.
```

```
two() -> 2.
```

```
add(X,Y) -> X() + Y().
```

Now open the Erlang shell, compile the module and get going:

```
1> c(hhfun).
```

```
{ok, hhfun}
```

```
2> hhfun:add(one,two).
```

```
** exception error: bad function one
```

```
    in function hhfun:add/2
```

```
3> hhfun:add(1,2).
```

```
** exception error: bad function 1
```

```
    in function hhfun:add/2
```

```
4> hhfun:add(fun hhfun:one/0, fun hhfun:two/0).
```

```
3
```

Confusing? Not so much, once you know how it works (isn't that always the case?) In command 2, the atoms `one` and `two` are passed to `add/2`, which then uses both atoms as function names (`X() + Y()`). If function names are written without a parameter list then those names are interpreted as atoms, and atoms can not be functions, so the call fails. This is the reason why expression 3 also fails: the values 1 and 2 can not be called as functions either, and functions are what we need!

This is why a new notation has to be added to the language in order to let you pass functions from outside a module. This is what `fun Module:Function/Arity is`: it tells the VM to use that specific function, and then bind it to a variable.

So what are the gains of using functions in that manner? Well a little example might be needed in order to understand it. We'll add

a few functions to hfuncs that work recursively over a list to add or subtract one from each integer of a list:

```
increment([]) -> [];
increment([H|T]) -> [H+1|increment(T)].
```

```
decrement([]) -> [];
decrement([H|T]) -> [H-1|decrement(T)].
```

See how similar these functions are? They basically do the same thing: they cycle through a list, apply a function on each element (+ or -) and then call themselves again. There is almost nothing changing in that code: only the applied function and the recursive call are different. The core of a recursive call on a list like that is always the same. We'll abstract all the similar parts in a single function (`map/2`) that will take another function as an argument:

```
map(_, []) -> [];
map(F, [H|T]) -> [F(H)|map(F,T)].
```

```
incr(X) -> X + 1.
decr(X) -> X - 1.
```

Which can then be tested in the shell:

```
1> c(hfuncs).
{ok, hfuncs}
2> L = [1,2,3,4,5].
[1,2,3,4,5]
3> hfuncs:increment(L).
[2,3,4,5,6]
4> hfuncs:decrement(L).
[0,1,2,3,4]
5> hfuncs:map(fun hfuncs:incr/1, L).
[2,3,4,5,6]
6> hfuncs:map(fun hfuncs:decr/1, L).
[0,1,2,3,4]
```

Here the results are the same, but you have just created a very smart abstraction! Every time you will want to apply a function to each element of a list, you only have to call map/2 with your function as a parameter. However, it is a bit annoying to have to put every function we want to pass as a parameter to map/2 in a module, name it, export it, then compile it, etc. In fact it's plainly unpractical. What we need are functions that can be declared on the fly...

Anonymous functions

Anonymous functions, or *funs*, address that problem by letting you declare a special kind of function inline, without naming them. They can do pretty much everything normal functions can do, except calling themselves recursively (how could they do it if they are anonymous?) Their syntax is:

```
fun(Args1) ->
    Expression1, Exp2, ..., ExpN;
(Args2) ->
    Expression1, Exp2, ..., ExpN;
(Args3) ->
    Expression1, Exp2, ..., ExpN
end
```

And can be used the following way:

```
7> Fn = fun() -> a end.
#Fun<erl_eval.20.67289768>
8> Fn().
a
9> hfun:map(fun(X) -> X + 1 end, L).
[2,3,4,5,6]
10> hfun:map(fun(X) -> X - 1 end, L).
[0,1,2,3,4]
```

And now you're seeing one of the things that make people like functional programming so much: the ability to make abstractions on a very low level of code. Basic concepts such as looping can thus be ignored, letting you focus on what is done rather than how to do it.

Anonymous functions are already pretty dandy for such abstractions but they still have more hidden powers:

```
11> PrepareAlarm = fun(Room) ->
11>     io:format("Alarm set in ~s.~n",[Room]),
11>     fun() -> io:format("Alarm tripped in ~s! Call Batman!~n",[Room]) end
11> end.
#Fun<erl_eval.20.67289768>
12> AlarmReady = PrepareAlarm("bathroom").
Alarm set in bathroom.
#Fun<erl_eval.6.13229925>
13> AlarmReady().
Alarm tripped in bathroom! Call Batman!
ok
```

Hold the phone Batman! What's going on here? Well, first of all, we declare an anonymous function assigned to *PrepareAlarm*. This function has not run yet: it only gets executed when *PrepareAlarm*("bathroom"). is called. At that point, the call to *io:format/2* is evaluated and the "Alarm set" text is output. The second expression (another anonymous function) is returned to the caller and then assigned to *AlarmReady*. Note that in this function, the variable *Room*'s value is taken from the 'parent' function (*PrepareAlarm*). This is related to a concept called *closures*.



To understand closures, one must first understand scope. A function's scope can be imagined as the place where all the variables and their values are stored. In the function `base(A) -> B = A + 1,` *A* and *B* are both defined to be part of `base/1`'s scope. This means that anywhere inside `base/1`, you can refer to *A* and *B* and expect a value to be bound to them. And when I say 'anywhere', I ain't kidding, kid; this includes anonymous functions too:

```
base(A) ->
  B = A + 1,
  F = fun() -> A * B end,
  F().
```

B and *A* are still bound to `base/1`'s scope, so the function *F* can still access them. This is because *F* inherits `base/1`'s scope. Like most kinds of real-life inheritance, the parents can't get what the children have:

```
base(A) ->
  B = A + 1,
  F = fun() -> C = A * B end,
  F(),
  C.
```

In this version of the function, *B* is still equal to *A + 1* and *F* will still execute fine. However, the variable *C* is only in the scope of the

anonymous function in *F*. When *base/1* tries to access *C*'s value on the last line, it only finds an unbound variable. In fact, had you tried to compile this function, the compiler would have thrown a fit. Inheritance only goes one way.

It is important to note that the inherited scope follows the anonymous function wherever it is, even when it is passed to another function:

```
a() ->
  Secret = "pony",
  fun() -> Secret end.
```

```
b(F) ->
  "a/0's password is "++F().
```

Then if we compile it:

```
14> c(hhfun).
{ok, hhfun}
15> hhfun:b(hhfun:a()).
"a/0's password is pony"
```

Who told *a/0*'s password? Well, *a/0* did. While the anonymous function has *a/0*'s scope when it's declared in there, it can still carry it when executed in *b/1*, as explained above. This is very useful because it lets us carry around parameters and content out of its original context, where the whole context itself are not needed anymore (exactly like we did with Batman in a previous example).

You're most likely to use anonymous functions to carry state around when you have functions defined that take many arguments, but you have a constant one:

```
16> math:pow(5,2).
```

```
25.0
```

```
17> Base = 2.  
2  
18> PowerOfTwo = fun(X) -> math:pow(Base,X) end.  
#Fun<erl_eval.6.13229925>  
17> hhfuns:map(PowerOfTwo, [1,2,3,4]).  
[2.0,4.0,8.0,16.0]
```

By wrapping the call to `math:pow/2` inside an anonymous function with the `Base` variable bound in its scope, we made it possible to have each of the calls to `PowerOfTwo` in `hhfuns:map/2` use the integers from the list as the exponents of our base.

A little trap you might fall into when writing anonymous functions is when you try to redefine the scope:

```
base() ->  
  A = 1,  
  (fun() -> A = 2 end)().
```

This will declare an anonymous function and then run it. As the anonymous function inherits `base/0`'s scope, trying to use the `=` operator compares 2 with the variable `A` (bound to 1). This is guaranteed to fail. However it is possible to redefine the variable if it's done in the nested function's head:

```
base() ->  
  A = 1,  
  (fun(A) -> A = 2 end)(2).
```

And this works. If you try to compile it, you'll get a warning about *shadowing* ("Warning: variable 'A' shadowed in 'fun'"). Shadowing is the term used to describe the act of defining a new variable that has the same name as one that was in the parent scope. This is there to prevent some mistakes (usually rightly so), so you might want to consider renaming your variables in these circumstances.

Update:

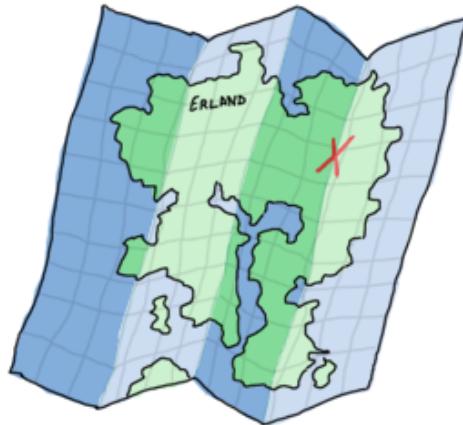
Starting with version 17.0, the language supports using anonymous functions with an internal name. That's right, *anonymous but named functions*.

The trick is that the name is visible only within the function's scope, not outside of it. The main advantage of this is that it makes it possible to define anonymous recursive functions. For example, we could make an anonymous function that keeps being loud forever:

```
18> f(PrepareAlarm), f(AlarmReady).
ok
19> PrepareAlarm = fun(Room) ->
19>   io:format("Alarm set in ~s.~n",[Room]),
19>   fun Loop() ->
19>     io:format("Alarm tripped in ~s! Call Batman!~n",[Room]),
19>     timer:sleep(500),
19>     Loop()
19>   end
19> end.
#Fun<erl_eval.6.71889879>
20> AlarmReady = PrepareAlarm("bathroom").
Alarm set in bathroom.
#Fun<erl_eval.44.71889879>
21> AlarmReady().
Alarm tripped in bathroom! Call Batman!
Alarm tripped in bathroom! Call Batman!
Alarm tripped in bathroom! Call Batman!
...
...
```

The *Loop* variable refers to the anonymous function itself, and within that scope, will be usable as any other similar variable pointing to an anonymous function. This should generally make a lot of operations in the shell a lot less painful moving on forward.

We'll set the anonymous function theory aside a bit and we'll explore more common abstractions to avoid having to write more recursive functions, like I promised at the end of the previous chapter.



Maps, filters, folds and more

At the beginning of this chapter, I briefly showed how to abstract away two similar functions to get a `map/2` function. I also affirmed that such a function could be used for any list where we want to act on each element. The function was the following:

```
map(_, []) -> [];
map(F, [H|T]) -> [F(H)|map(F,T)].
```

However, there are many other similar abstractions to build from commonly occurring recursive functions. Let's first take a look at these two functions:

```
%% only keep even numbers
even(L) -> lists:reverse(even(L,[])).
```

```
even([], Acc) -> Acc;
even([H|T], Acc) when H rem 2 == 0 ->
    even(T, [H|Acc]);
even([_|T], Acc) ->
```

```
even(T, Acc).
```

```
%% only keep men older than 60
old_men(L) -> lists:reverse(old_men(L,[])).
```

```
old_men([], Acc) -> Acc;
old_men([Person = {male, Age}|People], Acc) when Age > 60 ->
    old_men(People, [Person|Acc]);
old_men([_|People], Acc) ->
    old_men(People, Acc).
```

The first one takes a list of numbers and returns only those that are even. The second one goes through a list of people of the form `{Gender, Age}` and only keeps those that are males over 60. The similarities are a bit harder to find here, but we've got some common points. Both functions operate on lists and have the same objective of keeping elements that succeed some test (also a *predicate*) and then drop the others. From this generalization we can extract all the useful information we need and abstract them away:

```
filter(Pred, L) -> lists:reverse(filter(Pred, L,[])).
```

```
filter(_, [], Acc) -> Acc;
filter(Pred, [H|T], Acc) ->
    case Pred(H) of
        true -> filter(Pred, T, [H|Acc]);
        false -> filter(Pred, T, Acc)
    end.
```

To use the filtering function we now only need to get the test outside of the function. Compile the `hhfuns` module and try it:

```
1> c(hhfuns).
{ok, hhfuns}
2> Numbers = lists:seq(1,10).
```

```

[1,2,3,4,5,6,7,8,9,10]
3> hhfun:filter(fun(X) -> X rem 2 == 0 end, Numbers).
[2,4,6,8,10]
4> People = [{male,45},{female,67},{male,66},{female,12},{unknown,174},{male,74}].
[{male,45},{female,67},{male,66},{female,12},{unknown,174},{male,74}]
5> hhfun:filter(fun({Gender,Age}) -> Gender == male andalso Age > 60 end, People).
[{male,66},{male,74}]

```

These two examples show that with the use of the `filter/2` function, the programmer only has to worry about producing the predicate and the list. The act of cycling through the list to throw out unwanted items is no longer necessary to think about. This is one important thing about abstracting functional code: try to get rid of what's always the same and let the programmer supply in the parts that change.

In the previous chapter, another kind of recursive manipulation we applied on lists was to look at every element of a list one after the other and reduce them to a single answer. This is called a *fold* and can be used on the following functions:

```

%% find the maximum of a list
max([H|T]) -> max2(T, H).

```

```

max2([], Max) -> Max;
max2([H|T], Max) when H > Max -> max2(T, H);
max2([_|T], Max) -> max2(T, Max).

```

```

%% find the minimum of a list
min([H|T]) -> min2(T,H).

```

```

min2([], Min) -> Min;
min2([H|T], Min) when H < Min -> min2(T,H);
min2([_|T], Min) -> min2(T, Min).

```

```

%% sum of all the elements of a list

```

```
sum(L) -> sum(L,0).
```

```
sum([], sum) -> Sum;  
sum([H|T], sum) -> sum(T, H+Sum).
```



To find how the fold should behave, we've got to find all the common points of these actions and then what is different. As mentioned above, we always have a reduction from a list to a single value. Consequently, our fold should only consider iterating while keeping a single item, no list-building needed. Then we need to ignore the guards, because they're not always there: these need to be in the user's function. In this regard, our folding function will probably look a lot like sum.

A subtle element of all three functions that wasn't mentioned yet is that every function needs to have an initial value to start counting with. In the case of `sum/2`, we use `0` as we're doing addition and given $x = x + 0$, the value is neutral and we can't mess up the calculation by starting there. If we were doing multiplication we'd use `1` given $x = x * 1$. The functions `min/1` and `max/1` can't have a default starting value: if the list was only negative numbers and we started at `0`, the answer would be wrong. As such, we need to use the first element of the list as a starting point. Sadly, we can't always decide

this way, so we'll leave that decision to the programmer. By taking all these elements, we can build the following abstraction:

```
fold(_, Start, []) -> Start;
fold(F, Start, [H|T]) -> fold(F, F(H,Start), T).
```

And when tried:

```
6> c(hhfun).
{ok, hhfun}
7> [H|T] = [1,7,3,5,9,0,2,3].
[1,7,3,5,9,0,2,3]
8> hhfun:fold(fun(A,B) when A > B -> A; (_,B) -> B end, H, T).
9
9> hhfun:fold(fun(A,B) when A < B -> A; (_,B) -> B end, H, T).
0
10> hhfun:fold(fun(A,B) -> A + B end, 0, lists:seq(1,6)).
21
```

Pretty much any function you can think of that reduces lists to 1 element can be expressed as a fold.

What's funny there is that you can represent an accumulator as a single element (or a single variable), and an accumulator can be a list. Therefore, we can use a fold to build a list. This means fold is universal in the sense that you can implement pretty much any other recursive function on lists with a fold, even map and filter:

```
reverse(L) ->
  fold(fun(X,Acc) -> [X|Acc] end, [], L).

map2(F,L) ->
  reverse(fold(fun(X,Acc) -> [F(X)|Acc] end, [], L)).

filter2(Pred, L) ->
  F = fun(X,Acc) ->
    case Pred(X) of
```

```
    true -> [X|Acc];
    false -> Acc
end
end,
reverse(fold(F, [], L)).
```

And they all work the same as those written by hand before. How's that for powerful abstractions?

Map, filters and folds are only one of many abstractions over lists provided by the Erlang standard library (see `lists:map/2`, `lists:filter/2`, `lists:foldl/3` and `lists:foldr/3`). Other functions include `all/2` and `any/2` which both take a predicate and test if all the elements return true or if at least one of them returns true, respectively. Then you have `dropwhile/2` that will ignore elements of a list until it finds one that fit a certain predicate, its opposite, `takewhile/2`, that will keep all elements until there is one that doesn't return true to the predicate. A complimentary function to the two previous ones is `partition/2`, which will take a list and return two: one that has the terms which satisfy a given predicate, and one list for the others. Other frequently used lists functions include `flatten/1`, `flatlength/1`, `flatmap/2`, `merge/1`, `nth/2`, `nthtail/2`, `split/2` and a bunch of others.

You'll also find other functions such as zippers (as seen in last chapter), unzippers, combinations of maps and folds, etc. I encourage you to read the documentation on lists to see what can be done. You'll find yourself rarely needing to write recursive functions by using what's already been abstracted away by smart people.

Errors and Exceptions

Not so fast!



There's no right place for a chapter like this one. By now, you've learned enough that you're probably running into errors, but not yet enough to know how to handle them. In fact we won't be able to see all the error-handling mechanisms within this chapter. That's a bit because Erlang has two main paradigms: functional and concurrent. The functional subset is the one I've been explaining since the beginning of the book: referential transparency, recursion, higher order functions, etc. The concurrent subset is the one that makes Erlang famous: actors, thousands and thousands of concurrent processes, supervision trees, etc.

Because I judge the functional part essential to know before moving on to the concurrent part, I'll only cover the functional subset of the language in this chapter. If we are to manage errors, we must first understand them.

Note: Although Erlang includes a few ways to handle errors in functional code, most of the time you'll be told to just let it crash. I hinted at this in the [Introduction](#). The mechanisms that let you program this way are in the concurrent part of the language.

A Compilation of Errors

There are many kinds of errors: compile-time errors, logical errors, run-time errors and generated errors. I'll focus on compile-time errors in this section and go through the others in the next sections.

Compile-time errors are often syntactic mistakes: check your function names, the tokens in the language (brackets, parentheses, periods, commas), the arity of your functions, etc. Here's a list of some of the common compile-time error messages and potential resolutions in case you encounter them:

`module.beam: Module name 'madule' does not match file name 'module'`

The module name you've entered in the `-module` attribute doesn't match the filename.
`./module.erl:2: Warning: function some_function/0 is unused`

You have not exported a function, or the place where it's used has the wrong name or arity. It's also possible that you've written a function that is no longer needed. Check your code!

`./module.erl:2: function some_function/1 undefined`

The function does not exist. You've written the wrong name or arity either in the `-export` attribute or when declaring the function. This error is also output when the given function could not be compiled, usually because of a syntax error like forgetting to end a function with a period.

`./module.erl:5: syntax error before: 'SomeCharacterOrWord'`

This happens for a variety of reasons, namely unclosed parentheses, tuples or wrong expression termination (like closing the last branch of a `case` with a comma). Other reasons might include the use of a reserved atom in your code or unicode characters getting weirdly converted between different encodings (I've seen it happen!)

`./module.erl:5: syntax error before:`

All right, that one is certainly not as descriptive! This usually comes up when your line termination is not correct. This is a specific case of the previous error, so just keep an eye out.

`./module.erl:5: Warning: this expression will fail with a 'badarith' exception`

Erlang is all about dynamic typing, but remember that the types are strong. In this case, the compiler is smart enough to find that one of your arithmetic expressions will fail (say, `llama + 5`). It won't find type errors much more complex than that, though.

`./module.erl:5: Warning: variable 'Var' is unused`

You declared a variable and never use it afterwards. This might be a bug with your code, so double-check what you have written. Otherwise, you might want to switch the variable name to `_` or just prefix it with an underscore (something like `_Var`) if you feel the name helps make the code readable.

`./module.erl:5: Warning: a term is constructed, but never used`

In one of your functions, you're doing something such as building a list, declaring a tuple or an anonymous function without ever binding it to a variable or returning it. This warning tells you you're doing something useless or that you have made some mistake.

`./module.erl:5: head mismatch`

It's possible your function has more than one head, and each of them has a different arity. Don't forget that different arity means different functions, and you can't interleave function declarations that way. This error is also raised when you insert a function definition between the head clauses of another function.

`./module.erl:5: Warning: this clause cannot match because a previous clause at line 4 always matches`

A function defined in the module has a specific clause defined after a catch-all one. As such, the compiler can warn you that you'll never even need to go to the other branch.

`./module.erl:9: variable 'A' unsafe in 'case' (line 5)`

You're using a variable declared within one of the branches of a `case` ... of outside of it.

This is considered unsafe. If you want to use such variables, you'd be better off doing

`MyVar = case ... of...`

This should cover most errors you get at compile-time at this point. There aren't too many and most of the time the hardest part is finding which error caused a huge cascade of errors listed against other functions. It is better to resolve compiler errors in the order they were reported to avoid being misled by errors which may not actually be errors at all. Other kinds of errors sometimes appear and if you've got one I haven't included, send me an email and I'll add it along with an explanation as soon as possible.

No, YOUR logic is wrong!



Logical errors are the hardest kind of errors to find and debug. They're most likely errors coming from the programmer: branches of conditional statements such as 'if's and 'case's that don't consider all the cases, mixing up a multiplication for a division, etc. They do not make your programs crash but just end up giving you unseen bad data or having your program work in an unintended manner.

You're most likely on your own when it comes to this, but Erlang has many facilities to help you there, including test frameworks, TypEr and Dialyzer (as described in the [types chapter](#)), a debugger and tracing module, etc. Testing your code is likely your best defense. Sadly, there are enough of these kinds of errors in every programmer's career to write a few dozen books about so I'll avoid spending too much time here. It's easier to focus on those that make your programs crash, because it happens right there and won't bubble up 50 levels from now. Note that this is pretty much the origin of the 'let it crash' ideal I mentioned a few times already.

Run-time Errors

Run-time errors are pretty destructive in the sense that they crash your code. While Erlang has ways to deal with them, recognizing these errors is always helpful. As such, I've made a little list of common run-time errors with an explanation and example code that could generate them.

function_clause

```
1> lists:sort([3,2,1]).  
[1,2,3]  
2> lists:sort(fffffff).  
** exception error: no function clause matching lists:sort(fffffff)
```

All the guard clauses of a function failed, or none of the function clauses' patterns matched.

case_clause

```
3> case "Unexpected Value" of
3>   expected_value -> ok;
3>   other_expected_value -> 'also ok'
3> end.
** exception error: no case clause matching "Unexpected Value"
```

Looks like someone has forgotten a specific pattern in their case, sent in the wrong kind of data, or needed a catch-all clause!

if_clause

```
4> if 2 > 4 -> ok;
4>   0 > 1 -> ok
4> end.
** exception error: no true branch found when evaluating an if expression
```

This is pretty similar to case_clause errors: it can not find a branch that evaluates to true.

Ensuring you consider all cases or add the catch-all true clause might be what you need.

badmatch

```
5> [X,Y] = {4,5}.
** exception error: no match of right hand side value {4,5}
```

Badmatch errors happen whenever pattern matching fails. This most likely means you're trying to do impossible pattern matches (such as above), trying to bind a variable for the second time, or just anything that isn't equal on both sides of the = operator (which is pretty much what makes rebinding a variable fail!). Note that this error sometimes happens because the programmer believes that a variable of the form _MyVar is the same as _. Variables with an underscore are normal variables, except the compiler won't complain if they're not used. It is not possible to bind them more than once.

badarg

```
6> erlang:binary_to_list("heh, already a list").
** exception error: bad argument
  in function  binary_to_list/1
    called as binary_to_list("heh, already a list")
```

This one is really similar to function_clause as it's about calling functions with incorrect arguments. The main difference here is that this error is usually triggered by the programmer after validating the arguments from within the function, outside of the guard clauses. I'll show how to throw such errors later in this chapter.

undef

```
7> lists:random([1,2,3]).  
** exception error: undefined function lists:random/1
```

This happens when you call a function that doesn't exist. Make sure the function is exported from the module with the right arity (if you're calling it from outside the module) and double check that you did type the name of the function and the name of the module correctly. Another reason to get the message is when the module is not in Erlang's search path. By default, Erlang's search path is set to be in the current directory. You can add paths by using `code:add_patha/1` or `code:add_pathz/1`. If this still doesn't work, make sure you compiled the module to begin with!

badarith

```
8> 5 + llama.  
** exception error: bad argument in an arithmetic expression  
  in operator +/2  
  called as 5 + llama
```

This happens when you try to do arithmetic that doesn't exist, like divisions by zero or between atoms and numbers.

badfun

```
9> hhfuns:add(one,two).  
** exception error: bad function one  
in function hhfuns:add/2
```

The most frequent reason why this error occurs is when you use variables as functions, but the variable's value is not a function. In the example above, I'm using the `hhfuns` function from the [previous chapter](#) and using two atoms as functions. This doesn't work and `badfun` is thrown.

badarity

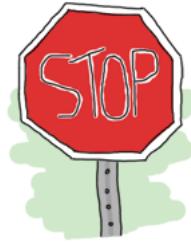
```
10> F = fun(_) -> ok end.  
#Fun<erl_eval.6.13229925>  
11> F(a,b).  
** exception error: interpreted function with arity 1 called with two arguments
```

The `badarity` error is a specific case of `badfun`: it happens when you use higher order functions, but you pass them more (or fewer) arguments than they can handle.

system_limit

There are many reasons why a `system_limit` error can be thrown: too many processes (we'll get there), atoms that are too long, too many arguments in a function, number of atoms too large, too many nodes connected, etc. To get a full list in details, read the Erlang Efficiency Guide on system limits. Note that some of these errors are serious enough to crash the whole VM.

Raising Exceptions



In trying to monitor the execution of code and protect against logical errors, it's often a good idea to provoke run-time crashes so problems will be spotted early.

There are three kinds of exceptions in Erlang: *errors*, *throws* and *exits*. They all have different uses (kind of):

Errors

Calling `erlang:error(Reason)` will end the execution in the current process and include a stack trace of the last functions called with their arguments when you catch it. These are the kind of exceptions that provoke the run-time errors above.

Errors are the means for a function to stop its execution when you can't expect the calling code to handle what just happened. If you get an `if_clause` error, what can you do? Change the code and recompile, that's what you can do (other than just displaying a pretty error message). An example of when not to use errors could be our tree module from the [recursion chapter](#). That module might not always be able to find a specific key in a tree when doing a lookup. In this case, it makes sense to expect the user to deal with unknown results: they could use a default value, check to insert a new one, delete the tree, etc. This is when it's appropriate to return a tuple of the form `{ok, Value}` or an atom like `undefined` rather than raising errors.

Now, errors aren't limited to the examples above. You can define your own kind of errors too:

```
1> erlang:error(badarith).
** exception error: bad argument in an arithmetic expression
2> erlang:error(custom_error).
** exception error: custom_error
```

Here, `custom_error` is not recognized by the Erlang shell and it has no custom translation such as "bad argument in ...", but it's usable in the same way and can be handled by the programmer in an identical manner (we'll see how to do that soon).

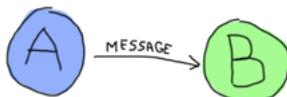
Exits

There are two kinds of exits: 'internal' exits and 'external' exits. Internal exits are triggered by calling the function `exit/1` and make the current process stop its execution. External exits are

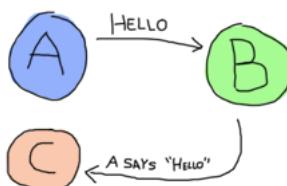
called with `exit/2` and have to do with multiple processes in the concurrent aspect of Erlang; as such, we'll mainly focus on internal exits and will visit the external kind later on.

Internal exits are pretty similar to errors. In fact, historically speaking, they were the same and only `exit/1` existed. They've got roughly the same use cases. So how to choose one? Well the choice is not obvious. To understand when to use one or the other, there's no choice but to start looking at the concepts of actors and processes from far away.

In the introduction, I've compared processes as people communicating by mail. There's not a lot to add to the analogy, so I'll go to diagrams and bubbles.



Processes here can send each other messages. A process can also listen for messages, wait for them. You can also choose what messages to listen to, discard some, ignore others, give up listening after a certain time etc.



These basic concepts let the implementors of Erlang use a special kind of message to communicate exceptions between processes. They act a bit like a process' last breath; they're sent right before a process dies and the code it contains stops executing. Other processes that were listening for that specific kind of message can then know about the event and do whatever they please with it. This includes logging, restarting the process that died, etc.



With this concept explained, the difference in using `erlang:error/1` and `exit/1` is easier to understand. While both can be used in an extremely similar manner, the real difference is in the intent. You can then decide whether what you've got is 'simply' an error or a condition worthy of killing the current process. This point is made stronger by the fact that `erlang:error/1` returns a stack trace and `exit/1` doesn't. If you were to have a pretty large stack trace or lots of arguments to the current function, copying the exit message to every listening process would mean copying the data. In some cases, this could become unpractical.

Throws

A throw is a class of exceptions used for cases that the programmer can be expected to handle. In comparison with exits and errors, they don't really carry any 'crash that process!' intent behind them, but rather control flow. As you use throws while expecting the programmer to handle them, it's usually a good idea to document their use within a module using them.

The syntax to throw an exception is:

```
1> throw(permission_denied).  
** exception throw: permission_denied
```

Where you can replace `permission_denied` by anything you want (including 'everything is fine', but that is not helpful and you will lose friends).

Throws can also be used for non-local returns when in deep recursion. An example of that is the `ssl` module which uses `throw/1` as a way to push `{error, Reason}` tuples back to a top-level function. This function then simply returns that tuple to the user. This lets the implementer only write for the successful cases and have one function deal with the exceptions on top of it all.

Another example could be the `array` module, where there is a `lookup` function that can return a user-supplied default value if it can't find the element needed. When the element can't be found, the value `default` is thrown as an exception, and the top-level function handles that and substitutes it with the user-supplied default value. This keeps the programmer of the module from needing to pass the default value as a parameter of every function of the `lookup` algorithm, again focusing only on the successful cases.



As a rule of thumb, try to limit the use of your throws for non-local returns to a single module in order to make it easier to debug your code. It will also let you change the innards of your module without requiring changes in its interface.

Dealing with Exceptions

I've already mentioned quite a few times that throws, errors and exits can be handled. The way to do this is by using a `try ... catch` expression.

A try ... catch is a way to evaluate an expression while letting you handle the successful case as well as the errors encountered. The general syntax for such an expression is:

```
try Expression of
  SuccessfulPattern1 [Guards] ->
    Expression1;
  SuccessfulPattern2 [Guards] ->
    Expression2
catch
  TypeOfError:ExceptionPattern1 ->
    Expression3;
  TypeOfError:ExceptionPattern2 ->
    Expression4
end.
```

The *Expression* in between try and of is said to be *protected*. This means that any kind of exception happening within that call will be caught. The patterns and expressions in between the try ... of and catch behave in exactly the same manner as a case ... of. Finally, the catch part: here, you can replace *TypeOfError* by either error, throw or exit, for each respective type we've seen in this chapter. If no type is provided, a throw is assumed. So let's put this in practice.

First of all, let's start a module named exceptions. We're going for simple here:

```
-module(exceptions).
-compile(export_all).

throws(F) ->
  try F() of
    _ -> ok
  catch
    Throw -> {throw, caught, Throw}
  end.
```

We can compile it and try it with different kinds of exceptions:

```
1> c(exceptions).
{ok,exceptions}
2> exceptions:throws(fun() -> throw(thrown) end).
{throw,caught,thrown}
3> exceptions:throws(fun() -> erlang:error(pang) end).
** exception error: pang
```

As you can see, this try ... catch is only receiving throws. As stated earlier, this is because when no type is mentioned, a throw is assumed. Then we have functions with catch clauses of each type:

```
errors(F) ->
  try F() of
    _ -> ok
```

```

catch
  error:Error -> {error, caught, Error}
end.

```

```

exits(F) ->
try F() of
  _ -> ok
catch
  exit:Exit -> {exit, caught, Exit}
end.

```

And to try them:

```

4> c(exceptions).
{ok,exceptions}
5> exceptions:errors(fun() -> erlang:error("Die!") end).
{error,caught,"Die!"}
6> exceptions:exits(fun() -> exit(goodbye) end).
{exit,caught,goodbye}

```

The next example on the menu shows how to combine all the types of exceptions in a single try ... catch. We'll first declare a function to generate all the exceptions we need:

```

sword(1) -> throw(slice);
sword(2) -> erlang:error(cut_arm);
sword(3) -> exit(cut_leg);
sword(4) -> throw(punch);
sword(5) -> exit(cross_bridge).

```

```

black_knight(Attack) when is_function(Attack, 0) ->
try Attack() of
  _ -> "None shall pass."
catch
  throw:slice -> "It is but a scratch.";
  error:cut_arm -> "I've had worse.";
  exit:cut_leg -> "Come on you pansy!";
  _:_ -> "Just a flesh wound."
end.

```

Here `is_function/2` is a BIF which makes sure the variable `Attack` is a function of arity 0. Then we add this one for good measure:

```
talk() -> "blah blah".
```

And now for something completely different:

```

7> c(exceptions).
{ok,exceptions}
8> exceptions:talk().
"blah blah"
9> exceptions:black_knight(fun exceptions:talk/0).
"None shall pass."

```

```

10> exceptions:black_knight(fun() -> exceptions:sword(1) end).
"It is but a scratch."
11> exceptions:black_knight(fun() -> exceptions:sword(2) end).
"I've had worse."
12> exceptions:black_knight(fun() -> exceptions:sword(3) end).
"Come on you pansy!"
13> exceptions:black_knight(fun() -> exceptions:sword(4) end).
"Just a flesh wound."
14> exceptions:black_knight(fun() -> exceptions:sword(5) end).
"Just a flesh wound."

```



The expression on line 9 demonstrates normal behavior for the black knight, when function execution happens normally. Each line that follows that one demonstrates pattern matching on exceptions according to their class (throw, error, exit) and the reason associated with them (slice, cut_arm, cut_leg).

One thing shown here on expressions 13 and 14 is a catch-all clause for exceptions. The `_` pattern is what you need to use to make sure to catch any exception of any type. In practice, you should be careful when using the catch-all patterns: try to protect your code from what you can handle, but not any more than that. Erlang has other facilities in place to take care of the rest.

There's also an additional clause that can be added after a `try ... catch` that will always be executed. This is equivalent to the 'finally' block in many other languages:

```

try Expr of
    Pattern -> Expr1
catch
    Type:Exception -> Expr2
after % this always gets executed
    Expr3
end

```

No matter if there are errors or not, the expressions inside the `after` part are guaranteed to run. However, you can not get any return value out of the `after` construct. Therefore, `after` is mostly used to run code with side effects. The canonical use of this is when you want to make sure a file you were reading gets closed whether exceptions are raised or not.

We now know how to handle the 3 classes of exceptions in Erlang with catch blocks. However, I've hidden information from you: it's actually possible to have more than one expression between the try and the of!

```
whoa() ->
    try
        talk(),
        _Knight = "None shall Pass!",
        _Doubles = [N*2 || N <- lists:seq(1,100)],
        throw(up),
        _WillReturnThis = tequila
    of
        tequila -> "hey this worked!"
    catch
        Exception:Reason -> {caught, Exception, Reason}
    end.
```

By calling `exceptions:whoa()`, we'll get the obvious {caught, throw, up}, because of `throw(up)`. So yeah, it's possible to have more than one expression between try and of...

What I just highlighted in `exceptions:whoa/0` and that you might have not noticed is that when we use many expressions in that manner, we might not always care about what the return value is. The of part thus becomes a bit useless. Well good news, you can just give it up:

```
im_impressed() ->
    try
        talk(),
        _Knight = "None shall Pass!",
        _Doubles = [N*2 || N <- lists:seq(1,100)],
        throw(up),
        _WillReturnThis = tequila
    catch
        Exception:Reason -> {caught, Exception, Reason}
    end.
```

And now it's a bit leaner!

Note: It is important to know that the protected part of an exception can't be tail recursive. The VM must always keep a reference there in case there's an exception popping up.

Because the try ... catch construct without the of part has nothing but a protected part, calling a recursive function from there might be dangerous for programs supposed to run for a long time (which is Erlang's niche). After enough iterations, you'll go out of memory or your program will get slower without really knowing why. By putting your recursive calls between the of and catch, you are not in a protected part and you will benefit from Last Call Optimisation.

Some people use try ... of ... catch rather than try ... catch by default to avoid unexpected errors of that kind, except for obviously non-recursive code with results that won't be used by

anything. You're most likely able to make your own decision on what to do!

Wait, there's more!

As if it wasn't enough to be on par with most languages already, Erlang's got yet another error handling structure. That structure is defined as the keyword `catch` and basically captures all types of exceptions on top of the good results. It's a bit of a weird one because it displays a different representation of exceptions:

```
1> catch throw(whoa).
whoa
2> catch exit(die).
{'EXIT',die}
3> catch 1/0.
{'EXIT',{badarith,[{erlang,'/',[1,0]}, {erl_eval,do_apply,5}, {erl_eval,expr,5}, {shell,exprs,6}, {shell,eval_exprs,6}, {shell,eval_loop,3}]}}
4> catch 2+2.
4
```

What we can see from this is that throws remain the same, but that exits and errors are both represented as `{'EXIT', Reason}`. That's due to errors being bolted to the language after exits (they kept a similar representation for backwards compatibility).

The way to read this stack trace is as follows:

```
5> catch doesnt:exist(a,4).
{'EXIT',{undef,[{doesnt,exist,[a,4]}, {erl_eval,do_apply,5}, {erl_eval,expr,5}, {shell,exprs,6}, {shell,eval_exprs,6}, {shell,eval_loop,3}]}}
```

- The type of error is `undef`, which means the function you called is not defined (see the list at the beginning of this chapter)
- The list right after the type of error is a stack trace
- The tuple on top of the stack trace represents the last function to be called (`{Module, Function, Arguments}`). That's your undefined function.
- The tuples after that are the functions called before the error. This time they're of the form `{Module, Function, Arity}`.
- That's all there is to it, really.

You can also manually get a stack trace by calling `erlang:get_stacktrace/0` in the process that crashed.

You'll often see `catch` written in the following manner (we're still in `exceptions.erl`):

```
catcher(X,Y) ->
    case catch X/Y of
        {'EXIT', {badarith,_}} -> "uh oh";
        N -> N
    end.
```

And as expected:

```
6> c(exceptions).
{ok,exceptions}
7> exceptions:catcher(3,3).
1.0
8> exceptions:catcher(6,3).
2.0
9> exceptions:catcher(6,0).
"uh oh"
```

This sounds compact and easy to catch exceptions, but there are a few problems with `catch`. The first of it is operator precedence:

```
10> X = catch 4+2.
* 1: syntax error before: 'catch'
10> X = (catch 4+2).
6
```

That's not exactly intuitive given that most expressions do not need to be wrapped in parentheses this way. Another problem with `catch` is that you can't see the difference between what looks like the underlying representation of an exception and a real exception:

```
11> catch erlang:boat().
{'EXIT',{undef,[{erlang,boat,[]}],
        [{erl_eval,do_apply,5},
         {erl_eval,expr,5},
         {shell,exprs,6},
         {shell,eval_exprs,6},
         {shell,eval_loop,3}]}}
```

```
12> catch exit({undef,[{erlang,boat,[]}],
                [{erl_eval,do_apply,5},
                 {erl_eval,expr,5},
                 {shell,exprs,6},
                 {shell,eval_exprs,6},
                 {shell,eval_loop,3}]})
```

And you can't know the difference between an error and an actual exit. You could also have used `throw/1` to generate the above exception. In fact, a `throw/1` in a `catch` might also be problematic in another scenario:

```
one_or_two(1) -> return;
one_or_two(2) -> throw(return).
```

And now the killer problem:

```
13> c(exceptions).
{ok,exceptions}
14> catch exceptions:one_or_two(1).
return
15> catch exceptions:one_or_two(2).
return
```

Because we're behind a `catch`, we can never know if the function threw an exception or if it returned an actual value! This might not really happen a whole lot in practice, but it's still a wart big enough to have warranted the addition of the `try ... catch` construct in the R10B release.

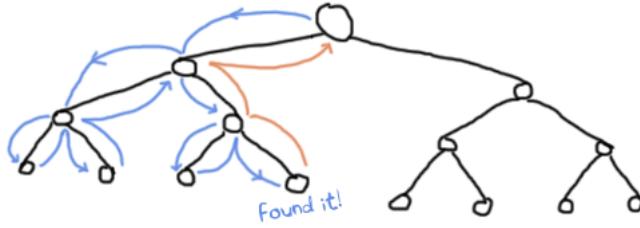
Try a try in a tree

To put exceptions in practice, we'll do a little exercise requiring us to dig for our `tree` module. We're going to add a function that lets us do a lookup in the tree to find out whether a value is already present in there or not. Because the tree is ordered by its keys and in this case we do not care about the keys, we'll need to traverse the whole thing until we find the value.

The traversal of the tree will be roughly similar to what we did in `tree:lookup/2`, except this time we will always search down both the left branch and the right branch. To write the function, you'll just need to remember that a tree node is either `{node, {Key, Value, NodeLeft, NodeRight}}` or `{node, 'nil'}` when empty. With this in hand, we can write a basic implementation without exceptions:

```
%% looks for a given value 'Val' in the tree.
has_value(_, {node, 'nil'}) ->
    false;
has_value(Val, {node, {_, Val, _, _}}) ->
    true;
has_value(Val, {node, {_, _, Left, Right}}) ->
    case has_value(Val, Left) of
        true -> true;
        false -> has_value(Val, Right)
    end.
```

The problem with this implementation is that every node of the tree we branch at has to test for the result of the previous branch:



This is a bit annoying. With the help of throws, we can make something that will require less comparisons:

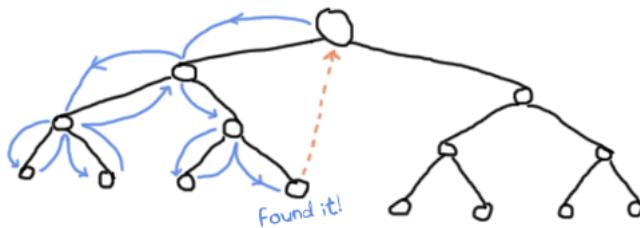
```

has_value(Val, Tree) ->
    try has_value1(Val, Tree) of
        false -> false
    catch
        true -> true
    end.

has_value1(_, {node, 'nil'}) ->
    false;
has_value1(Val, {node, {_, Val, _, _}}) ->
    throw(true);
has_value1(Val, {node, {_, _, Left, Right}}) ->
    has_value1(Val, Left),
    has_value1(Val, Right).

```

The execution of the code above is similar to the previous version, except that we never need to check for the return value: we don't care about it at all. In this version, only a throw means the value was found. When this happens, the tree evaluation stops and it falls back to the catch on top. Otherwise, the execution keeps going until the last false is returned and that's what the user sees:



Of course, the implementation above is longer than the previous one. However, it is possible to realize gains in speed and in clarity by using non-local returns with a throw, depending on the operations you're doing. The current example is a simple comparison and there's not much to see, but the practice still makes sense with more complex data structures and operations.

That being said, we're probably ready to solve real problems in sequential Erlang.

Functionally Solving Problems

Sounds like we're ready to do something practical with all that Erlang juice we drank. Nothing new is going to be shown but how to apply bits of what we've seen before. The problems in this chapter were taken from Miran's Learn You a Haskell. I decided to take the same solutions so curious readers can compare solutions in Erlang and Haskell as they wish. If you do so, you might find the final results to be pretty similar for two languages with such different syntaxes. This is because once you know functional concepts, they're relatively easy to carry over to other functional languages.

Reverse Polish Notation Calculator

Most people have learned to write arithmetic expressions with the operators in-between the numbers $((2 + 2) / 5)$. This is how most calculators let you insert mathematical expressions and probably the notation you were taught to count with in school. This notation has the downside of needing you to know about operator precedence: multiplication and division are more important (have a higher precedence) than addition and subtraction.

Another notation exists, called *prefix notation* or *Polish notation*, where the operator comes before the operands. Under this notation, $(2 + 2) / 5$ would become $(/ (+ 2 2) 5)$. If we decide to say `+` and `/` always take two arguments, then $(/ (+ 2 2) 5)$ can simply be written as `/ + 2 2 5`.

However, we will instead focus on *Reverse Polish notation* (or just *RPN*), which is the opposite of prefix notation: the operator follows the operands. The same example as above in RPN would be written `2 2 + 5 /`. Other example expressions could be `9 * 5 + 7` or `10 * 2 * (3 + 4) / 2` which get translated to `9 5 * 7 +` and `10 2 * 3 4 + * 2 /`, respectively. This notation was used a whole lot in early models of calculators as it would take little

memory to use. In fact some people still carry RPN calculators around. We'll write one of these.

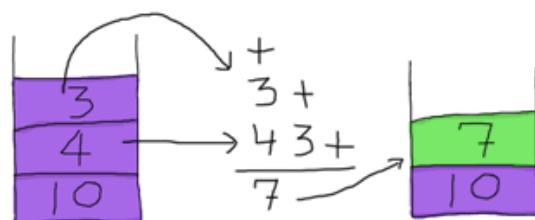
First of all, it might be good to understand how to read RPN expressions. One way to do it is to find the operators one by one and then regroup them with their operands by arity:

```
10 4 3 + 2 * -  
10 (4 3 +) 2 * -  
10 ((4 3 +) 2 *) -  
(10 ((4 3 +) 2 *)) -  
(10 (7 2 *)) -  
(10 14 -)  
-4
```

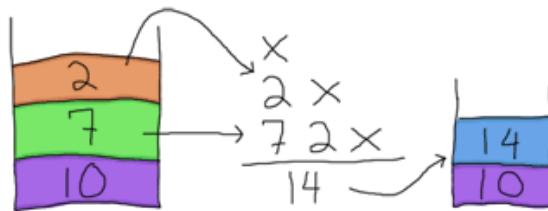
However, in the context of a computer or a calculator, a simpler way to do it is to make a *stack* of all the operands as we see them. Taking the mathematical expression $10 4 3 + 2 * -$, the first operand we see is 10. We add that to the stack. Then there's 4, so we also push that on top of the stack. In third place, we have 3; let's push that one on the stack too. Our stack should now look like this:



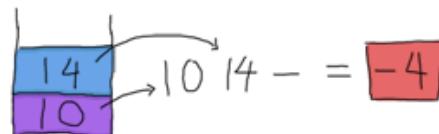
The next character to parse is a +. That one is a function of arity 2. In order to use it we will need to feed it two operands, which will be taken from the stack:



So we take that 7 and push it back on top of the stack (yuck, we don't want to keep these filthy numbers floating around!) The stack is now [7,10] and what's left of the expression is $2 * -$. We can take the 2 and push it on top of the stack. We then see *, which needs two operands to work. Again, we take them from the stack:



And push 14 back on top of our stack. All that remains is $-$, which also needs two operands. O Glorious luck! There are two operands left in our stack. Use them!



And so we have our result. This stack-based approach is relatively fool-proof and the low amount of parsing needed to be done before starting to calculate results explains why it was a good idea for old calculators to use this. There are other reasons to use RPN, but this is a bit out of the scope of this guide, so you might want to hit the Wikipedia article instead.

Writing this solution in Erlang is not too hard once we've done the complex stuff. It turns out the tough part is figuring out what steps need to be done in order to get our end result and we just did that. Neat. Open a file named `calc.erl`.

The first part to worry about is how we're going to represent a mathematical expression. To make things simple, we'll probably input them as a string: "10 4 3 + 2 * -". This string has whitespace, which isn't

part of our problem-solving process, but is necessary in order to use a simple tokenizer. What would be usable then is a list of terms of the form `["10","4","3","+","2","*","-"]` after going through the tokenizer. Turns out the function `string:tokens/2` does just that:

```
1> string:tokens("10 4 3 + 2 * -", " ").
[\"10\", \"4\", \"3\", \"+\", \"2\", \"*\", \"-\"]
```

That will be a good representation for our expression. The next part to define is the stack. How are we going to do that? You might have noticed that Erlang's lists act a lot like a stack. Using the `cons ()` operator in `[Head|Tail]` effectively behaves the same as pushing *Head* on top of a stack (*Tail*, in this case). Using a list for a stack will be good enough.

To read the expression, we just have to do the same as we did when solving the problem by hand. Read each value from the expression, if it's a number, put it on the stack. If it's a function, pop all the values it needs from the stack, then push the result back in. To generalize, all we need to do is go over the whole expression as a loop only once and accumulate the results. Sounds like the perfect job for a fold!

What we need to plan for is the function that `lists:foldl/3` will apply on every operator and operand of the expression. This function, because it will be run in a fold, will need to take two arguments: the first one will be the element of the expression to work with and the second one will be the stack.

We can start writing our code in the `calc.erl` file. We'll write the function responsible for all the looping and also the removal of spaces in the expression:

```
-module(calc).
-export([rpn/1]).
```

```
rpn(L) when is_list(L) ->
```

```
[Res] = lists:foldl(fun rpn/2, [], string:tokens(L, " ")),  
    Res.
```

We'll implement `rpn/2` next. Note that because each operator and operand from the expression ends up being put on top of the stack, the solved expression's result will be on that stack. We need to get that last value out of there before returning it to the user. This is why we pattern match over `[Res]` and only return `Res`.

Alright, now to the harder part. Our `rpn/2` function will need to handle the stack for all values passed to it. The head of the function will probably look like `rpn(Op,Stack)` and its return value like `[NewVal|Stack]`. When we get regular numbers, the operation will be:

```
rpn(X, Stack) -> [read(X)|Stack].
```

Here, `read/1` is a function that converts a string to an integer or floating point value. Sadly, there is no built-in function to do this in Erlang (only one or the other). We'll add it ourselves:

```
read(N) ->  
  case string:to_float(N) of  
    {error,no_float} -> list_to_integer(N);  
    {F,_} -> F  
  end.
```

Where `string:to_float/1` does the conversion from a string such as "13.37" to its numeric equivalent. However, if there is no way to read a floating point value, it returns `{error,no_float}`. When that happens, we need to call `list_to_integer/1` instead.

Now back to `rpn/2`. The numbers we encounter all get added to the stack. However, because our pattern matches on anything (see [Pattern Matching](#)), operators will also get pushed on the stack. To avoid this, we'll put them all in preceding clauses. The first one we'll try this with is the addition:

```
rpn("+", [N1,N2|S]) -> [N2+N1|S];
rpn(X, Stack) -> [read(X)|Stack].
```

We can see that whenever we encounter the "+" string, we take two numbers from the top of the stack ($N1, N2$) and add them before pushing the result back onto that stack. This is exactly the same logic we applied when solving the problem by hand. Trying the program we can see that it works:

```
1> c(calc).
{ok,calc}
2> calc:rpn("3 5 +").
8
3> calc:rpn("7 3 + 5 +").
15
```

The rest is trivial, as you just need to add all the other operators:

```
rpn("+", [N1,N2|S]) -> [N2+N1|S];
rpn("-", [N1,N2|S]) -> [N2-N1|S];
rpn("*", [N1,N2|S]) -> [N2*N1|S];
rpn("/", [N1,N2|S]) -> [N2/N1|S];
rpn("^", [N1,N2|S]) -> [math:pow(N2,N1)|S];
rpn("ln", [N|S]) -> [math:log(N)|S];
rpn("log10", [N|S]) -> [math:log10(N)|S];
rpn(X, Stack) -> [read(X)|Stack].
```

Note that functions that take only one argument such as logarithms only need to pop one element from the stack. It is left as an exercise to the reader to add functions such as 'sum' or 'prod' which return the sum of all the elements read so far or the products of them all. To help you out, they are implemented in my version of calc.erl already.

To make sure this all works fine, we'll write very simple unit tests. Erlang's = operator can act as an *assertion* function. Assertions should crash whenever they encounter unexpected values, which is exactly what we need. Of course, there are more advanced testing

frameworks for Erlang, including Common Test and EUnit. We'll check them out later, but for now the basic = will do the job:

```
rpn_test() ->
    5 = rpn("2 3 +"),
    87 = rpn("90 3 -"),
    -4 = rpn("10 4 3 + 2 * -"),
    -2.0 = rpn("10 4 3 + 2 * - 2 /"),
    ok = try
        rpn("90 34 12 33 55 66 + * - +")
    catch
        error:{badmatch,[_|_]} -> ok
    end,
    4037 = rpn("90 34 12 33 55 66 + * - + -"),
    8.0 = rpn("2 3 ^"),
    true = math:sqrt(2) == rpn("2 0.5 ^"),
    true = math:log(2.7) == rpn("2.7 ln"),
    true = math:log10(2.7) == rpn("2.7 log10"),
    50 = rpn("10 10 10 20 sum"),
    10.0 = rpn("10 10 10 20 sum 5 /"),
    1000.0 = rpn("10 10 20 0.5 prod"),
    ok.
```

The test function tries all operations; if there's no exception raised, the tests are considered successful. The first four tests check that the basic arithmetic functions work right. The fifth test specifies behaviour I have not explained yet. The try ... catch expects a badmatch error to be thrown because the expression can't work:

```
90 34 12 33 55 66 + * - +
90 (34 (12 (33 (55 66 +) *) -) +)
```

At the end of `rpn/1`, the values -3947 and 90 are left on the stack because there is no operator to work on the 90 that hangs there. Two ways to handle this problem are possible: either ignore it and only take the value on top of the stack (which would be the last result calculated) or crash because the arithmetic is wrong. Given Erlang's policy is to let it crash, it's what was chosen here. The part that actually crashes is the

[Res] in rpn/1. That one makes sure only one element, the result, is left in the stack.

The few tests that are of the form true = FunctionCall1 == FunctionCall2 are there because you can't have a function call on the left hand side of =. It still works like an assert because we compare the comparison's result to true.

I've also added the test cases for the sum and prod operators so you can exercise yourselves implementing them. If all tests are successful, you should see the following:

```
1> c(calc).
{ok,calc}
2> calc:rpn_test().
ok
3> calc:rpn("1 2 ^ 2 2 ^ 3 2 ^ 4 2 ^ sum 2 -").
28.0
```

Where 28 is indeed equal to $\sum(1^2 + 2^2 + 3^2 + 4^2) - 2$. Try as many of them as you wish.

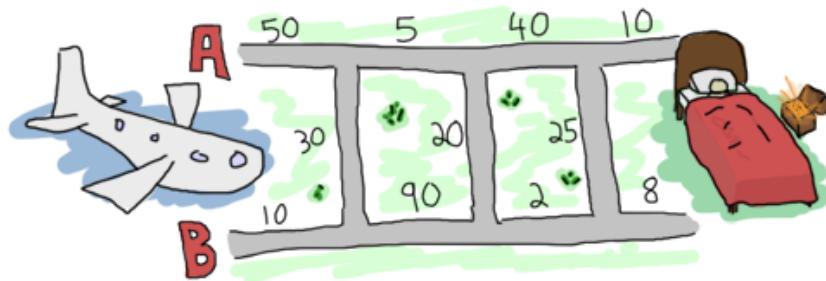
One thing that could be done to make our calculator better would be to make sure it raises badarith errors when it crashes because of unknown operators or values left on the stack, rather than our current badmatch error. It would certainly make debugging easier for the user of the calc module.

Heathrow to London

Our next problem is also taken from Learn You a Haskell. You're on a plane due to land at Heathrow airport in the next hours. You have to get to London as fast as possible; your rich uncle is dying and you want to be the first there to claim dibs on his estate.

There are two roads going from Heathrow to London and a bunch of smaller streets linking them together. Because of speed limits and

usual traffic, some parts of the roads and smaller streets take longer to drive on than others. Before you land, you decide to maximize your chances by finding the optimal path to his house. Here's the map you've found on your laptop:



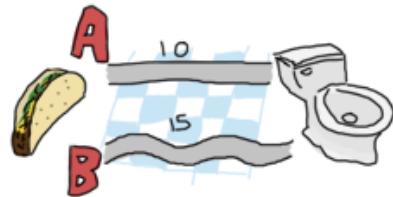
Having become a huge fan of Erlang after reading online books, you decide to solve the problem using that language. To make it easier to work with the map, you enter data the following way in a file named `road.txt`:

```
50
10
30
5
90
20
40
2
25
10
8
0
```

The road is laid in the pattern: A₁, B₁, X₁, A₂, B₂, X₂, ..., A_n, B_n, X_n, where X is one of the roads joining the A side to the B side of the map. We insert a 0 as the last X segment, because no matter what we do we're at our destination already. Data can probably be organized in tuples of 3 elements (triples) of the form {A,B,X}.

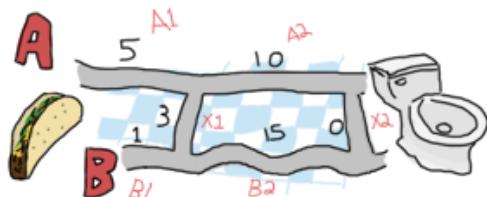
The next thing you realize is that it's worth nothing to try to solve this problem in Erlang when you don't know how to solve it by hand to begin with. In order to do this, we'll use what recursion taught us.

When writing a recursive function, the first thing to do is to find our base case. For our problem at hand, this would be if we had only one tuple to analyze, that is, if we only had to choose between A, B (and crossing X, which in this case is useless because we're at destination):



Then the choice is only between picking which of path A or path B is the shortest. If you've learned your recursion right, you know that we ought to try and converge towards the base case. This means that on each step we'll take, we'll want to reduce the problem to choosing between A and B for the next step.

Let's extend our map and start over:

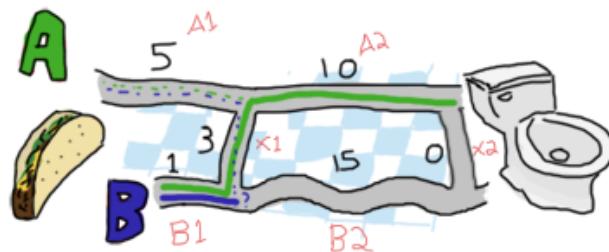


Ah! It gets interesting! How can we reduce the triple $\{5,1,3\}$ to a strict choice between A and B? Let's see how many options are possible for A. To get to the intersection of A_1 and A_2 (I'll call this the *point A1*), I can either take road A_1 directly (5), or come from B_1 (1) and then cross over X_1 (3). In this case, The first option (5) is longer than the second one (4). For the option A, the shortest path is [B, x]. So what are the options for

B? You can either proceed from $A1$ (5) then cross over $X1$ (3), or strictly take the path $B1$ (1).

Alright! What we've got is a length 4 with the path $[B, x]$ towards the first intersection A and a length 1 with the path $[B]$ towards the intersection of $B1$ and $B2$. We then have to decide what to pick between going to the second point A (the intersection of $A2$ and the endpoint or $X2$) and the second point B (intersection of $B2$ and $X2$). To make a decision, I suggest we do the same as before. Now you don't have much choice but to obey, given I'm the guy writing this text. Here we go!

All possible paths to take in this case can be found in the same way as the previous one. We can get to the next point A by either taking the path $A2$ from $[B, x]$, which gives us a length of 14 ($14 = 4 + 10$), or by taking $B2$ then $X2$ from $[B]$, which gives us a length of 16 ($16 = 1 + 15 + 0$). In this case, the path $[B, x, A]$ is better than $[B, B, x]$.



We can also get to the next point B by either taking the path $A2$ from $[B, x]$ and then crossing over $X2$ for a length of 14 ($14 = 4 + 10 + 0$), or by taking the road $B2$ from $[B]$ for a length of 16 ($16 = 1 + 15$). Here, the best path is to pick the first option, $[B, x, A, x]$.

So when this whole process is done, we're left with two paths, A or B, both of length 14. Either of them is the right one. The last selection will always have two paths of the same length, given the last X segment has a length 0. By solving our problem recursively, we've made sure to always get the shortest path at the end. Not too bad, eh?

Subtly enough, we've given ourselves the basic logical parts we need to build a recursive function. You can implement it if you want, but I promised we would have very few recursive functions to write ourselves. We'll use a fold.

Note: while I have shown folds being used and constructed with lists, folds represent a broader concept of iterating over a data structure with an accumulator. As such, folds can be implemented over trees, dictionaries, arrays, database tables, etc.

It is sometimes useful when experimenting to use abstractions like maps and folds; they make it easier to later change the data structure you use to work with your own logic.

So where were we? Ah, yes! We had the file we're going to feed as input ready. To do file manipulations, the file module is our best tool. It contains many functions common to many programming languages in order to deal with files themselves (setting permissions, moving files around, renaming and deleting them, etc.)

It also contains the usual functions to read and/or write from files such as: `file:open/2` and `file:close/1` to do as their names say (opening and closing files!), `file:read/2` to get the content a file (either as string or a binary), `file:read_line/1` to read a single line, `file:position/3` to move the pointer of an open file to a given position, etc.

There's a bunch of shortcut functions in there too, such as `file:read_file/1` (opens and reads the contents as a binary), `file:consult/1` (opens and parses a file as Erlang terms) or `file:pread/2` (changes a position and then reads) and `pwrite/2` (changes the position and writes content).

With all these choices available, it's going to be easy to find a function to read our `road.txt` file. Because we know our road is relatively small, we're going to call `file:read_file("road.txt")`:

```

1> {ok, Binary} = file:read_file("road.txt").
{ok,<<"50\r\n10\r\n30\r\n5\r\n90\r\n20\r\n40\r\n2\r\n25\r\n10\r\n8\r\n0\r\n">>}
2> S = string:tokens(binary_to_list(Binary), "\r\n\t").
["50","10","30","5","90","20","40","2","25","10","8","0"]

```

Note that in this case, I added a space (" ") and a tab ("\t") to the valid tokens so the file could have been written in the form "50 10 30 5 90 20 40 2 25 10 8 0" too. Given that list, we'll need to transform the strings into integers. We'll use a similar manner to what we used in our RPN calculator:

```

3> [list_to_integer(X) || X <- S].
[50,10,30,5,90,20,40,2,25,10,8,0]

```

Let's start a new module called road.erl and write this logic down:

```

-module(road).
-compile(export_all).

main() ->
    File = "road.txt",
    {ok, Bin} = file:read_file(File),
    parse_map(Bin).

parse_map(Bin) when is_binary(Bin) ->
    parse_map(binary_to_list(Bin));
parse_map(Str) when is_list(Str) ->
    [list_to_integer(X) || X <- string:tokens(Str, "\r\n\t")].

```

The function main/0 is here responsible for reading the content of the file and passing it on to parse_map/1. Because we use the function file:read_file/1 to get the contents out of road.txt, the result we obtain is a binary. For this reason, I've made the function parse_map/1 match on both lists and binaries. In the case of a binary, we just call the function again with the string being converted to a list (our function to split the string works on lists only.)

The next step in parsing the map would be to regroup the data into the {A,B,X} form described earlier. Sadly, there's no simple generic way to pull elements from a list 3 at a time, so we'll have to pattern match our way in a recursive function in order to do it:

```
group_vals([], Acc) ->
    lists:reverse(Acc);
group_vals([A,B,X|Rest], Acc) ->
    group_vals(Rest, [{A,B,X} | Acc]).
```

That function works in a standard tail-recursive manner; there's nothing too complex going on here. We'll just need to call it by modifying `parse_map/1` a bit:

```
parse_map(Bin) when is_binary(Bin) ->
    parse_map(binary_to_list(Bin));
parse_map(Str) when is_list(Str) ->
    Values = [list_to_integer(X) || X <- string:tokens(Str, "\r\n\t")],
    group_vals(Values, []).
```

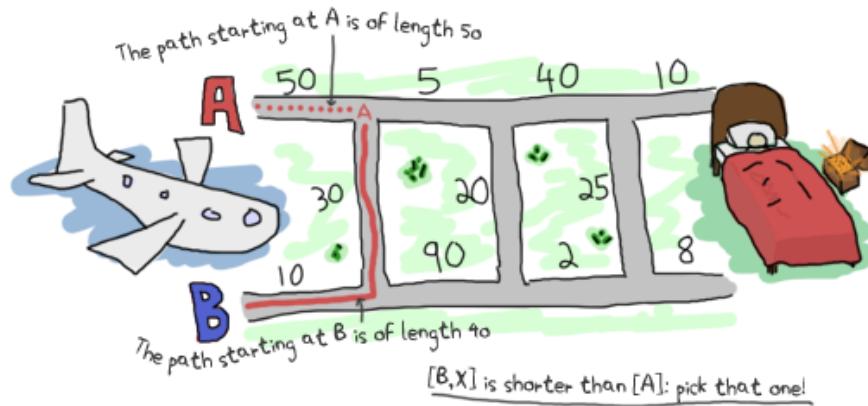
If we try and compile it all, we should now have a road that makes sense:

```
1> c(road).
{ok,road}
2> road:main().
[{50,10,30},{5,90,20},{40,2,25},{10,8,0}]
```

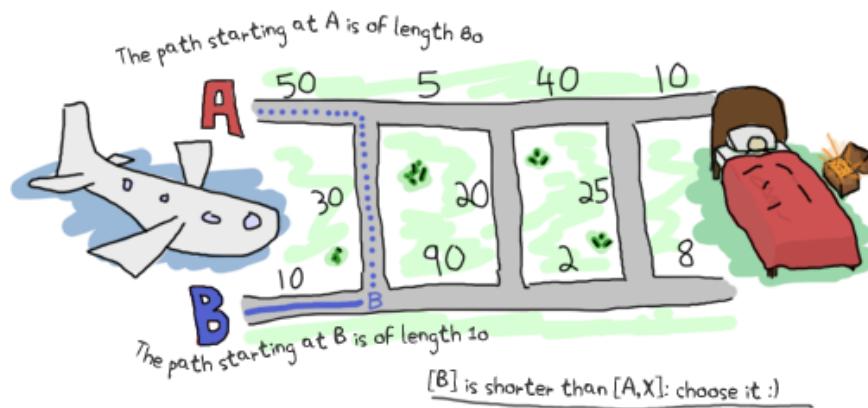
Ah yes, that looks right. We get the blocks we need to write our function that will then fit in a fold. For this to work, finding a good accumulator is necessary.

To decide what to use as an accumulator, the method I find the easiest to use is to imagine myself in the middle of the algorithm while it runs. For this specific problem, I'll imagine that I'm currently trying to find the shortest path of the second triple ({5,90,20}). To decide on which path is the best, I need to have the result from the previous triple.

Luckily, we know how to do it, because we don't need an accumulator and we got all that logic out already. So for A:



And take the shortest of these two paths. For B, it was similar:



So now we know that the current best path coming from A is [B, X]. We also know it has a length of 40. For B, the path is simply [B] and the length is 10. We can use this information to find the next best paths for A and B by reapplying the same logic, but counting the previous ones in the expression. The other data we need is the path traveled so we can show it to the user. Given we need two paths (one for A and one for B) and two accumulated lengths, our accumulator can take the form `{DistanceA, PathA}, {DistanceB, PathB}`. That way, each iteration of the fold has access to all the state and we build it up to show it to the user in the end.

This gives us all the parameters our function will need: the {A,B,X} triples and an accumulator of the form {{DistanceA,PathA}, {DistanceB,PathB}}.

Putting this into code in order to get our accumulator can be done the following way:

```
shortest_step({A,B,X}, {{DistA,PathA}, {DistB,PathB}}) ->
    OptA1 = {DistA + A, [{a,A}|PathA]},
    OptA2 = {DistB + B + X, [{x,X}, {b,B}|PathB]},
    OptB1 = {DistB + B, [{b,B}|PathB]},
    OptB2 = {DistA + A + X, [{x,X}, {a,A}|PathA]},
    {erlang:min(OptA1, OptA2), erlang:min(OptB1, OptB2)}.
```

Here, *OptA1* gets the first option for A (going through A), *OptA2* the second one (going through B then X). The variables *OptB1* and *OptB2* get the similar treatment for point B. Finally, we return the accumulator with the paths obtained.

About the paths saved in the code above, note that I decided to use the form [{x,x}] rather than [x] for the simple reason that it might be nice for the user to know the length of each segment. The other thing I'm doing is that I'm accumulating the paths backwards ({x,x} comes before {b,B}.) This is because we're in a fold, which is tail recursive: the whole list is reversed, so it is necessary to put the last one traversed before the others.

Finally, I use `erlang:min/2` to find the shortest path. It might sound weird to use such a comparison function on tuples, but remember that every Erlang term can be compared to any other! Because the length is the first element of the tuple, we can sort them that way.

What's left to do is to stick that function into a fold:

```
optimal_path(Map) ->
    {A,B} = lists:foldl(fun shortest_step/2, {{0,[ ]}, {0,[ ]}}, Map),
    {_Dist,Path} = if hd(element(2,A)) =/= {x,0} -> A;
                    hd(element(2,B)) =/= {x,0} -> B
```

```
    end,  
    lists:reverse(Path).
```

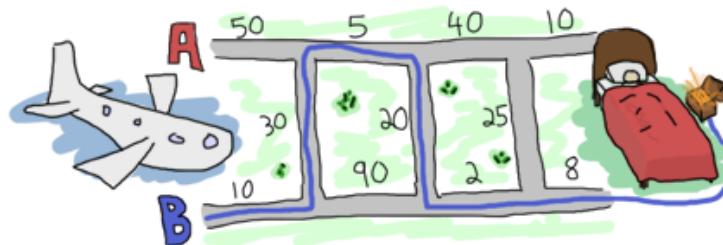
At the end of the fold, both paths should end up having the same distance, except one's going through the final $\{x,0\}$ segment. The if looks at the last visited element of both paths and returns the one that doesn't go through $\{x,0\}$. Picking the path with the fewest steps (compare with `length/1`) would also work. Once the shortest one has been selected, it is reversed (it was built in a tail-recursive manner; you **must** reverse it). You can then display it to the world, or keep it secret and get your rich uncle's estate. To do that, you have to modify the main function to call `optimal_path/1`. Then it can be compiled.

```
main() ->  
  File = "road.txt",  
  {ok, Bin} = file:read_file(File),  
  optimal_path(parse_map(Bin)).
```

Oh, look! We've got the right answer! Great Job!

```
1> c(road).  
{ok,road}  
2> road:main().  
[{b,10},{x,30},{a,5},{x,20},{b,2},{b,8}]
```

Or, to put it in a visual way:



But you know what would be really useful? Being able to run our program from outside the Erlang shell. We'll need to change our main function again:

```

main([FileName]) ->
{ok, Bin} = file:read_file(FileName),
Map = parse_map(Bin),
io:format("~p~n",[optimal_path(Map)]),
erlang:halt().

```

The main function now has an arity of 1, needed to receive parameters from the command line. I've also added the function `erlang:halt/0`, which will shut down the Erlang VM after being called. I've also wrapped the call to `optimal_path/1` into `io:format/2` because that's the only way to have the text visible outside the Erlang shell.

With all of this, your `road.erl` file should now look like this (minus comments):

```

-module(road).
-compile(export_all).

main([FileName]) ->
{ok, Bin} = file:read_file(FileName),
Map = parse_map(Bin),
io:format("~p~n",[optimal_path(Map)]),
erlang:halt(0).

%% Transform a string into a readable map of triples
parse_map(Bin) when is_binary(Bin) ->
parse_map(binary_to_list(Bin));
parse_map(Str) when is_list(Str) ->
Values = [list_to_integer(X) || X <- string:tokens(Str,"\\r\\n\\t")],
group_vals(Values, []).

group_vals([], Acc) ->
lists:reverse(Acc);
group_vals([A,B,X|Rest], Acc) ->
group_vals(Rest, [{A,B,X} | Acc]).

%% Picks the best of all paths, woo!
optimal_path(Map) ->
{A,B} = lists:foldl(fun shortest_step/2, {{0,[ ]}, {0,[ ]}}, Map),

```

```

{_Dist,Path} = if hd(element(2,A))=/={x,0} -> A;
               hd(element(2,B))=/={x,0} -> B
               end,
lists:reverse(Path).

%% actual problem solving
%% change triples of the form {A,B,X}
%% where A,B,X are distances and a,b,x are possible paths
%% to the form {DistanceSum, PathList}.
shortest_step({A,B,X}, {{DistA,PathA}, {DistB,PathB}}) ->
    OptA1 = {DistA + A, [{a,A}|PathA]},
    OptA2 = {DistB + B + X, [{x,X}, {b,B}|PathB]},
    OptB1 = {DistB + B, [{b,B}|PathB]},
    OptB2 = {DistA + A + X, [{x,X}, {a,A}|PathA]},
    {erlang:min(OptA1, OptA2), erlang:min(OptB1, OptB2)}.

```

And running the code:

```

$ erlc road.erl
$ erl -noshell -run road main road.txt
[{b,10},{x,30},{a,5},{x,20},{b,2},{b,8}]

```

And yep, it's right! It's pretty much all you need to do to get things to work. You could make yourself a bash/batch file to wrap the line into a single executable, or you could check out escript to get similar results.

As we've seen with these two exercises, solving problems is much easier when you break them off in small parts that you can solve individually before piecing everything together. It's also not worth much to go ahead and program something without understanding it. Finally, a few tests are always appreciated. They'll let you make sure everything works fine and will let you change the code without changing the results at the end.

A Short Visit to Common Data Structures

Won't be too long, promised!

Chances are you now understand the functional subset of Erlang pretty well and could read many programs without a problem. However, I could bet it's still a bit hard to think about how to build a real useful program even though the last chapter was about solving problems in a functional manner. I'm saying this because it's how I felt like at about that point in my learning, but if you're doing better, congratulations!

Anyway, the point I'm coming to is that we've seen a bunch of things: most basic data types, the shell, how to write modules and functions (with recursion), different ways to compile, control the flow of the program, handle exceptions, abstract away some common operations, etc. We've also seen how to store data with tuples, lists and an incomplete implementation of a binary search tree. What we haven't seen is the other data structures provided to the programmer in the Erlang standard library.



Records

Records are, first of all, a hack. They are more or less an afterthought to the language and can have their share of inconveniences. I'll cover that later. They're still pretty useful whenever you have a small data structure where you want to access the attributes by name directly. As such, Erlang records are a lot like structs in C (if you know C.)

They're declared as module attributes in the following manner:

```
-module(records).  
-compile(export_all).  
  
-record(robot, {name,  
               type=industrial,  
               hobbies,  
               details=[]}).
```

So here we have a record representing robots with 4 fields: name, type, hobbies and details. There is also a default value for type and details, industrial and [], respectively. Here's how to declare a record in the module records:

```
first_robot() ->  
    #robot{name="Mechatron",  
           type=handmade,  
           details=["Moved by a small man inside"]}.
```

And running the code:

```
1> c(records).  
{ok,records}  
2> records:first_robot().  
{robot,"Mechatron",handmade,undefined,  
 ["Moved by a small man inside"]}
```

Woops! Here comes the hack! Erlang records are just syntactic sugar on top of tuples. Fortunately, there's a way to make it better. The Erlang shell has a command `rr(Module)` that lets you load record definitions from *Module*:

```
3> rr(records).
[robot]
4> records:first_robot().
#robot{name = "Mechatron", type = handmade,
      hobbies = undefined,
      details = ["Moved by a small man inside"]}
```

Ah there! This makes it much easier to work with records that way. You'll notice that in `first_robot/0`, we had not defined the `hobbies` field and it had no default value in its declaration. Erlang, by defaults, sets the value to `undefined` for you.

To see the behavior of the defaults we set in the `robot` definition, let's compile the following function:

```
car_factory(CorpName) ->
    #robot{name=CorpName, hobbies="building cars"}.
```

And run it:

```
5> c(records).
{ok,records}
6> records:car_factory("Jokeswagen").
#robot{name = "Jokeswagen", type = industrial,
      hobbies = "building cars", details = []}
```

And we have an industrial robot that likes to spend time building cars.

Note: The function rr() can take more than a module name: it can take a wildcard (like rr("*")) and also a list as a second argument to specify which records to load.

There are a few other functions to deal with records in the shell: rd(Name, Definition) lets you define a record in a manner similar to the -record(Name, Definition) used in our module. You can use rf() to 'unload' all records, or rf(Name) or rf([Names]) to get rid of specific definitions.

You can use rl() to print all record definitions in a way you could copy-paste into the module or use rl(Name) or rl([Names]) to restrict it to specific records.

Finally, rp(Term) lets you convert a tuple to a record (given the definition exists).

Writing records alone won't do much. We need a way to extract values from them. There are basically two ways to do this. The first one is with a special 'dot syntax'. Assuming you have the record definition for robots loaded:

```
5> Crusher = #robot{name="Crusher", hobbies=["Crushing people","petting cats"]}.
#robot{name = "Crusher", type = industrial,
      hobbies = ["Crushing people","petting cats"],
      details = []}
6> Crusher#robot.hobbies.
["Crushing people","petting cats"]
```

Ugh, not a pretty syntax. This is due to the nature of records as tuples. Because they're just some kind of compiler trick, you have to keep keywords around defining what record goes with what variable, hence the #robot part of Crusher#robot.hobbies. It's sad, but

there's no way out of it. Worse than that, nested records get pretty ugly:

```
7> NestedBot = #robot{details=#robot{name="erNest"}}.  
#robot{name = undefined,type = industrial,  
hobbies = undefined,  
details = #robot{name = "erNest",type = industrial,  
hobbies = undefined,details = []}}  
8> (NestedBot#robot.details)#robot.name.  
"erNest"
```

And yes, the parentheses are mandatory.

Update:

Starting with revision R14A, it is now possible to nest records without the parentheses. The *NestedBot* example above could also be written as `NestedRobot#robot.details#robot.name` and work the same.

To further show the dependence of records on tuples, see the following:

```
9> #robot.type.  
3
```

What this outputs is which element of the underlying tuple it is.

One saving feature of records is the possibility to use them in function heads to pattern match and also in guards. Declare a new record as follows on top of the file, and then add the functions under:

```
-record(user, {id, name, group, age}).
```

```
%% use pattern matching to filter
```

```
admin_panel(#user{name=Name, group=admin}) ->
    Name ++ " is allowed!";
admin_panel(#user{name=Name}) ->
    Name ++ " is not allowed".
```

```
%% can extend user without problem
adult_section(U = #user{}) when U#user.age >= 18 ->
    %% Show stuff that can't be written in such a text
    allowed;
adult_section(_) ->
    %% redirect to sesame street site
    forbidden.
```

The syntax to bind a variable to any field of a record is demonstrated in the `admin_panel/1` function (it's possible to bind variables to more than one field). An important thing to note about the `adult_section/1` function is that you need to do `someVar = #some_record{}` in order to bind the whole record to a variable.

Then we do the compiling as usual:

```
10> c(records).
{ok,records}
11> rr(records).
[robot,user]
12> records:admin_panel(#user{id=1, name="ferd", group=admin, age=96}).
"ferd is allowed!"
13> records:admin_panel(#user{id=2, name="you", group=users, age=66}).
"you is not allowed"
14> records:adult_section(#user{id=21, name="Bill", group=users, age=72}).
allowed
15> records:adult_section(#user{id=22, name="Noah", group=users, age=13}).
forbidden
```

What this lets us see is how it is not necessary to match on all parts of the tuple or even know how many there are when writing

the function: we can only match on the age or the group if that's what's needed and forget about all the rest of the structure. If we were to use a normal tuple, the function definition might need to look a bit like `function({record, _, _, ICareAboutThis, _, _}) ->` Then, whenever someone decides to add an element to the tuple, someone else (probably angry about it all) would need to go around and update all functions where that tuple is used.

The following function illustrates how to update a record (they wouldn't be very useful otherwise):

```
repairman(Rob) ->
    Details = Rob#robot.details,
    NewRob = Rob#robot{details=["Repaired by repairman"|Details]},
    {repaired, NewRob}.
```

And then:

```
16> c(records).
{ok,records}
17> records:repairman(#robot{name="Ulbert", hobbies=["trying to have feelings"]}).
{repaired,#robot{name = "Ulbert",type = industrial,
    hobbies = ["trying to have feelings"],
    details = ["Repaired by repairman"]}}
```

And you can see my robot has been repaired. The syntax to update records is a bit special here. It looks like we're updating the record in place (`Rob#robot{Field=NewValue}`) but it's all compiler trickery to call the underlying `erlang:setelement/3` function.

One last thing about records. Because they're pretty useful and code duplication is annoying, Erlang programmers frequently share records across modules with the help of *header files*. Erlang header files are pretty similar to their C counter-part: they're

nothing but a snippet of code that gets added to the module as if it were written there in the first place. Create a file named records.hrl with the following content:

```
%% this is a .hrl (header) file.  
-record(included, {some_field,  
                  some_default = "yeah!",  
                  unimaginative_name}).
```

To include it in records.erl, just add the following line to the module:

```
-include("records.hrl").
```

And then the following function to try it:

```
included() -> #included{some_field="Some value"}.
```

Now, try it as usual:

```
18> c(records).  
{ok,records}  
19> rr(records).  
[included,robot,user]  
20> records:included().  
#included{some_field = "Some value",some_default = "yeah!",  
         unimaginative_name = undefined}
```

Hooray! That's about it for records; they're ugly but useful. Their syntax is not pretty, they're not much but a hack, but they're relatively important for the maintainability of your code.

Note: You will often see open source software using the method shown here of having a project-wide .hrl file for records that are shared across all modules. While I felt obligated to document this use, I strongly recommend that you keep all record definitions

local, within one module. If you want some other module to look at a record's innards, write functions to access its fields and keep its details as private as possible. This helps prevent name clashes, avoids problems when upgrading code, and just generally improves the readability and maintainability of your code.

Key-Value Stores



I've had you build a tree back a few chapters, and the use was to use it as a key-value store for an address book. That book sucked: we couldn't delete or convert it to anything useful. It was a good demonstration of recursion, but not much more. Now is the time to introduce you to a bunch of useful data structures and modules to store data under a certain key. I won't define what every function does nor go through all the modules. I will simply link to the doc pages. Consider me as someone responsible about 'raising awareness about key-value stores in Erlang' or something. Sounds like a good title. I just need one of these ribbons.

For small amounts of data, there are basically two data structures that can be used. The first one is called a *proplist*. A proplist is any list of tuples of the form [{Key,Value}]. They're a weird kind of structure because there is no other rule than that. In fact the rules are so relaxed that the list can also contain boolean values, integers and whatever you want. We're rather interested by the idea of a tuple with a key and a value in a list here, though. To work with prolists, you can use the prolists module. It

contains functions such as `prolists:delete/2`, `prolists:get_value/2`, `prolists:get_all_values/2`, `prolists:lookup/2` and `prolists:lookup_all/2`.

You'll notice there is no function to add or update an element of the list. This shows how loosely defined prolists are as a data structure. To get these functionalities, you must cons your element manually (`[NewElement|OldList]`) and use functions such as `lists:keyreplace/4`. Using two modules for one small data structure is not the cleanest thing, but because prolists are so loosely defined, they're often used to deal with configuration lists, and general description of a given item. Prolists are not exactly complete data structures. They're more of a common pattern that appears when using lists and tuples to represent some object or item; the prolists module is a bit of a toolbox over such a pattern.

If you do want a more complete key-value store for small amounts of data, the `orddict` module is what you need. Oreddicts (ordered dictionaries) are prolists with a taste for formality. Each key can be there once, the whole list is sorted for faster average lookup, etc. Common functions for the CRUD usage include `orddict:store/3`, `orddict:find/2` (when you do not know whether the key is in the dictionaries), `orddict:fetch/2` (when you know it is there or that it **must** be there) and `orddict:erase/2`.



Orddicts are a generally good compromise between complexity and efficiency up to about 75 elements (see my benchmark). After that amount, you should switch to different key-value stores.

There are basically two key-value structures/modules to deal with larger amounts of data: dicts and gb_trees. Dictionaries have the same interface as orddicts: dict:store/3, dict:find/2, dict:fetch/2, dict:erase/2 and every other function, such as dict:map/2 and dict:fold/2 (pretty useful to work on the whole data structure!) Dicts are thus very good choices to scale orddicts up whenever it is needed.

General Balanced Trees, on the other hand, have a bunch more functions leaving you more direct control over how the structure is to be used. There are basically two modes for gb_trees: the mode where you know your structure in and out (I call this the 'smart mode'), and the mode where you can't assume much about it (I call this one the 'naive mode'). In naive mode, the functions are gb_trees:enter/3, gb_trees:lookup/2 and gb_trees:delete_any/2. The related smart functions are gb_trees:insert/3, gb_trees:get/2, gb_trees:update/3 and gb_trees:delete/2. There is also gb_trees:map/2, which is always a nice thing when you need it.

The disadvantage of 'naive' functions over 'smart' ones is that because gb_trees are balanced trees, whenever you insert a new element (or delete a bunch), it might be possible that the tree will need to balance itself. This can take time and memory (even in useless checks just to make sure). The 'smart' function all assume that the key is present in the tree: this lets you skip all the safety checks and results in faster times.

When should you use gb_trees over dicts? Well, it's not a clear decision. As the benchmark module I have written will show, gb_trees and dicts have somewhat similar performances in many respects. However, the benchmark demonstrates that dicts have the best read speeds while the gb_trees tend to be a little quicker on other operations. You can judge based on your own needs which one would be the best.

Oh and also note that while dicts have a fold function, gb_trees don't: they instead have an *iterator* function, which returns a bit of the tree on which you can call gb_trees:next(*Iterator*) to get the following values in order. What this means is that you need to write your own recursive functions on top of gb_trees rather than use a generic fold. On the other hand, gb_trees let you have quick access to the smallest and largest elements of the structure with gb_trees:smallest/1 and gb_trees:largest/1.

I would therefore say that your application's needs is what should govern which key-value store to choose. Different factors such as how much data you've got to store, what you need to do with it and whatnot all have their importance. Measure, profile and benchmark to make sure.

Note: some special key-value stores exist to deal with resources of different size. Such stores are ETS tables, DETS tables and the mnesia database. However, their use is strongly related to the concepts of multiple processes and distribution. Because of this, they'll only be approached later on. I'm leaving this as a reference to pique your curiosity and for those interested.

Update:

Starting with version 17.0, the language supports a new native

key-value data type, described in [Postscript: Maps](#). They should be the new de-facto replacement for dicts.

Arrays

But what about code that requires data structures with nothing but numeric keys? Well for that, there are arrays. They allow you to access elements with numerical indices and to fold over the whole structure while possibly ignoring undefined slots.

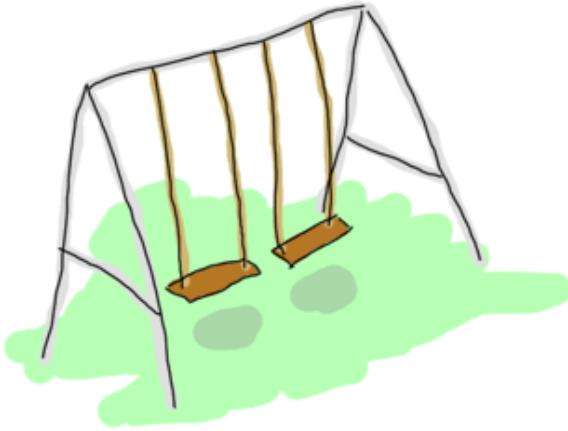
Don't drink too much kool-aid:

Erlang arrays, at the opposite of their imperative counterparts, are not able to have such things as constant-time insertion or lookup. Because they're usually slower than those in languages which support destructive assignment and that the style of programming done with Erlang doesn't necessarily lend itself too well to arrays and matrices, they are rarely used in practice.

Generally, Erlang programmers who need to do matrix manipulations and other uses requiring arrays tend to use concepts called Ports to let other languages do the heavy lifting, or C-Nodes, Linked in drivers and NIFs (Experimental, R13B03+).

Arrays are also weird in the sense that they're one of the few data structures to be 0-indexed (at the opposite of tuples or lists), along with indexing in the regular expressions module. Be careful with them.

A Set of Sets



If you've ever studied set theory in whatever mathematics class you have an idea about what sets can do. If you haven't, you might want to skip over this. However, I'll just say that sets are groups of unique elements that you can compare and operate on: find which elements are in two groups, in none of them, only in one or the other, etc. There are more advanced operations letting you define relations and operate on these relations and much more. I'm not going to dive into the theory (again, it's out of the scope of this book) so I'll just describe them as it is.

There are 4 main modules to deal with sets in Erlang. This is a bit weird at first, but it makes more sense once you realize that it's because it was agreed by implementers that there was no 'best' way to build a set. The four modules are `ordsets`, `sets`, `gb_sets` and `sofs` (sets of sets):

ordsets

Ordsets are implemented as a sorted list. They're mainly useful for small sets, are the slowest kind of set, but they have the simplest and most readable representation of all sets. There are standard functions for them such as `ordsets:new/0`,

`ordsets:is_element/2`, `ordsets:add_element/2`, `ordsets:del_element/2`,
`ordsets:union/1`, `ordsets:intersection/1`, and a bunch more.

sets

Sets (the module) is implemented on top of a structure really similar to the one used in `dict`. They implement the same interface as `ordsets`, but they're going to scale much better. Like dictionaries, they're especially good for read-intensive manipulations, like checking whether some element is part of the set or not.

gb_sets

`Gb_sets` themselves are constructed above a General Balanced Tree structure similar to the one used in the `gb_trees` module. `gb_sets` are to sets what `gb_tree` is to `dict`; an implementation that is faster when considering operations different than reading, leaving you with more control. While `gb_sets` implement the same interface as sets and `ordsets`, they also add more functions. Like `gb_trees`, you have smart vs. naive functions, iterators, quick access to the smallest and largest values, etc.

sofs

Sets of sets (`sofs`) are implemented with sorted lists, stuck inside a tuple with some metadata. They're the module to use if you want to have full control over relationships between sets, families, enforce set types, etc. They're really what you want if you need mathematics concept rather than 'just' groups of unique elements.

Don't drink too much kool-aid:

While such a variety can be seen as something great, some implementation details can be downright frustrating. As an example, `gb_sets`, `ordsets` and `sofs` all use the `==` operator to

compare values: if you have the numbers 2 and 2.0, they'll both end up seen as the same one.

However, sets (the module) uses the `=:=` operator, which means you can't necessarily switch over every implementation as you wish. There are cases where you need one precise behavior and at that point, you might lose the benefit of having multiple implementations.

It's a bit confusing to have that many options available. Björn Gustavsson, from the Erlang/OTP team and programmer of Wings3D mainly suggests using `gb_sets` in most circumstances, using `ordset` when you need a clear representation that you want to process with your own code and '`sets`' when you need the `=:=` operator (source.)

In any case, like for key-value stores, the best solution is usually to benchmark and see what fits your application better.

Directed Graphs

There is one other data structure that I want to mention here (not that there are not more than what's mentioned in this chapter, on the contrary): directed graphs. Again, this data structure is more for readers who already know the mathematical theory that goes with it.

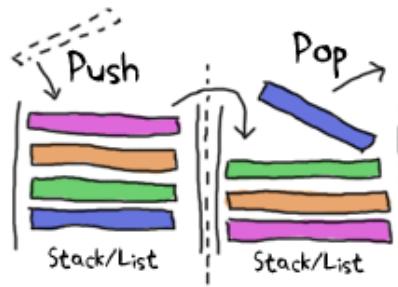
Directed graphs in Erlang are implemented as two modules, `digraph` and `digraph_utils`. The `digraph` module basically allows the construction and modification of a directed graph: manipulating edges and vertices, finding paths and cycles, etc. On the other hand, `digraph_utils` allows you to navigate a graph

(postorder, preorder), testing for cycles, arborescences or trees, finding neighbors, and so on.

Because directed graphs are closely related to set theory, the 'sofs' module contains a few functions letting you convert families to digraphs and digraphs to families.

Queues

The queue module implements a double-ended FIFO (First In, First Out) queue:



They're implemented a bit as illustrated above: two lists (in this context, stacks) that allow to both append and prepend elements rapidly.

The queue module basically has different functions in a mental separation into 3 interfaces (or APIs) of varying complexity, called 'Original API', 'Extended API' and 'Okasaki API':

Original API

The original API contains the functions at the base of the queue concept, including: `new/0`, for creating empty queues, `in/2`, for inserting new elements, `out/1`, for removing elements, and then functions to convert to lists, reverse the queue, look if a particular value is part of it, etc.

Extended API

The extended API mainly adds some introspection power and flexibility: it lets you do things such as looking at the front of the queue without removing the first element (see `get/1` or `peek/1`), removing elements without caring about them (`drop/1`), etc. These functions are not essential to the concept of queues, but they're still useful in general.

Okasaki API

The Okasaki API is a bit weird. It's derived from Chris Okasaki's *Purely Functional Data Structures*. The API provides operations similar to what was available in the two previous APIs, but some of the function names are written backwards and the whole thing is relatively peculiar. Unless you do know you want this API, I wouldn't bother with it.

You'll generally want to use queues when you'll need to ensure that the first item ordered is indeed the first one processed. So far, the examples I've shown mainly used lists as accumulators that would then be reversed. In cases where you can't just do all the reversing at once and elements are frequently added, the queue module is what you want (well, you should test and measure first! Always test and measure first!)

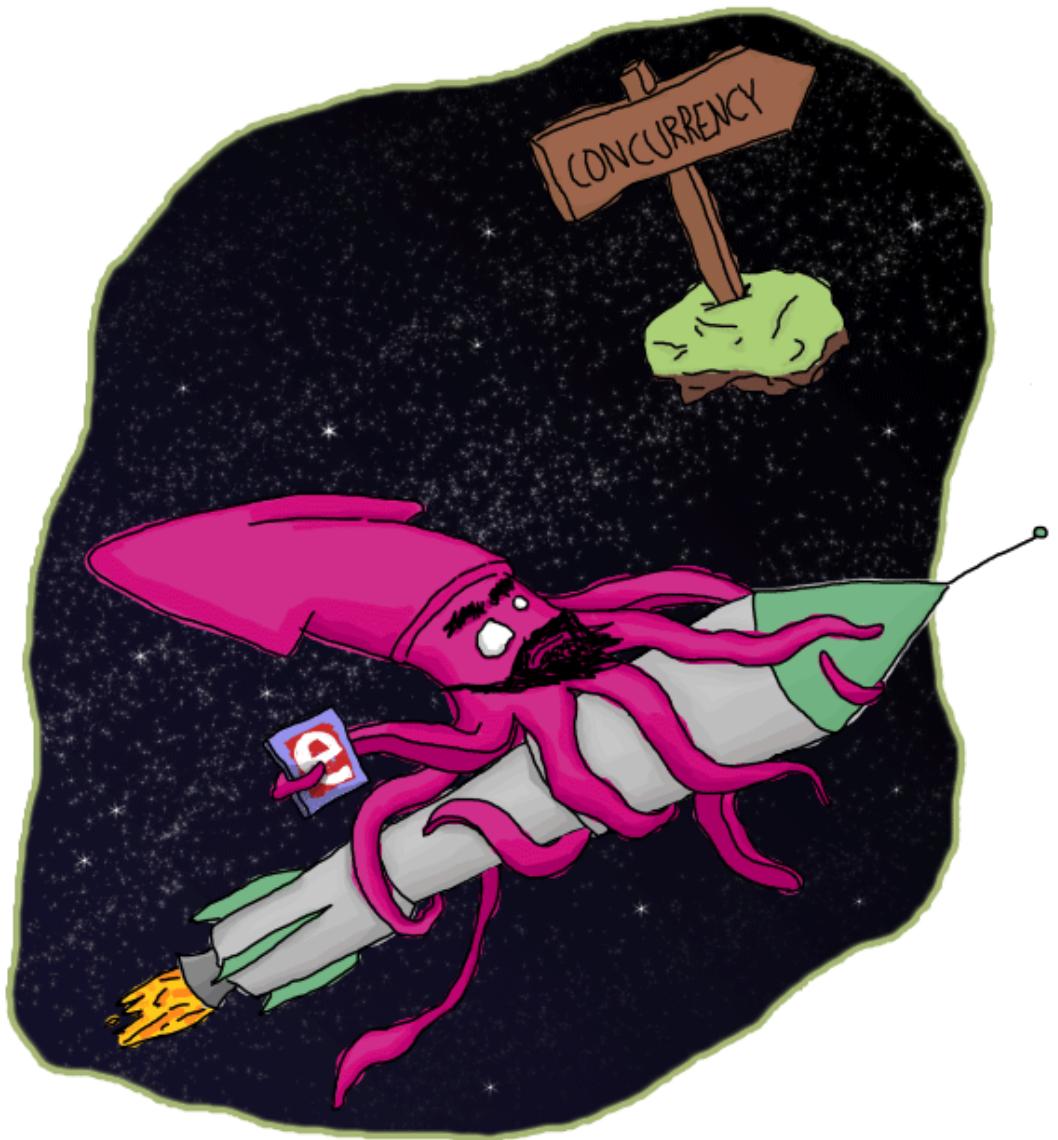
End of the short visit

That's about it for the data structures trip of Erlang. Thank you for having kept your arms inside the vehicles the whole time. Of course, there are a few more data structures available than that to solve different problems. I've only covered those that you're likely to encounter or need the most given the strengths of general use cases of Erlang. I encourage you to explore the

standard library and the extended one too to find more information.

You might be glad to learn that this completes our trip into sequential (functional) Erlang. I know a lot of people get in Erlang to see all the concurrency and processes and whatnot. It's understandable, given it's really where Erlang shines. Supervision trees, fancy error management, distribution, and more. I know I've been very impatient to write about these subjects, so I guess some readers were very impatient to read about them.

However, I judged it made more sense to be comfortable with functional Erlang before moving on to concurrent Erlang. It will be easier to move on afterwards and focus on all the new concepts. Here we go!



The Hitchhiker's Guide to Concurrency

Far out in the uncharted backwaters of the unfashionable beginning of the 21st century lies a small subset of human knowledge.

Within this subset of human knowledge is an utterly insignificant little discipline whose Von Neumann-descended architecture is so amazingly primitive that it is still thought that RPN calculators are a pretty neat idea.

This discipline has — or rather had — a problem, which was this: most of the people studying it were unhappy for pretty much of the time when trying to write parallel software. Many solutions were suggested for this problem, but most of these were largely concerned with the handling of little pieces of logic called locks and mutexes and whatnot, which is odd because on the whole it wasn't the small pieces of logic that needed parallelism.

And so the problem remained; lots of people were mean, and most of them were miserable, even those with RPN calculators.

Many were increasingly of the opinion that they'd all made a big mistake in trying to add parallelism to their programming languages, and that no program should have ever left its initial thread.

Note: parodying The Hitchhiker's Guide to the Galaxy is fun.
Read the book if you haven't already. It's good!

Don't Panic

Hi. Today (or whatever day you are reading this, even tomorrow), I'm going to tell you about concurrent Erlang. Chances are you've read about or dealt with concurrency before. You might also be curious about the emergence of multi-core programming. Anyway, the probabilities are high that you're reading this book because of all this talk about concurrency going on these days.



A warning though; this chapter is mostly theoretic. If you have a headache, a distaste for programming language history or just want to program, you might be better off skipping to the [end of the chapter](#) or skip to the next one (where more practical knowledge is shown.)

I've already explained in the book's intro that Erlang's concurrency was based on message passing and the actor model, with the example of people communicating with

nothing but letters. I'll explain it more in details again later, but first of all, I believe it is important to define the difference between *concurrency* and *parallelism*.

In many places both words refer to the same concept. They are often used as two different ideas in the context of Erlang. For many Erlangers, concurrency refers to the idea of having many actors running independently, but not necessarily all at the same time. Parallelism is having actors running exactly at the same time. I will say that there doesn't seem to be any consensus on such definitions around various areas of computer science, but I will use them in this manner in this text. Don't be surprised if other sources or people use the same terms to mean different things.

This is to say Erlang had concurrency from the beginning, even when everything was done on a single core processor in the '80s. Each Erlang process would have its own slice of time to run, much like desktop applications did before multi-core systems.

Parallelism was still possible back then; all you needed to do was to have a second computer running the code and communicating with the first one. Even then, only two actors could be run in parallel in this setup. Nowadays, multi-core systems allows for parallelism on a single computer (with some industrial chips having many dozens of cores) and Erlang takes full advantage of this possibility.

Don't drink too much Kool-Aid:

The distinction between concurrency and parallelism is important to make, because many programmers hold the belief that Erlang was ready for multi-core computers years before it actually was. Erlang was only adapted to true symmetric multiprocessing in the mid 2000s and only got most of the implementation right with the R13B release of the language in 2009. Before that, SMP often had to be disabled to avoid performance losses. To get parallelism on a multicore computer without SMP, you'd start many instances of the VM instead.

An interesting fact is that because Erlang concurrency is all about isolated processes, it took no conceptual change at the language level to bring true parallelism to the language. All the changes were transparently done in the VM, away from the eyes of the programmers.

Concepts of Concurrency



Back in the day, Erlang's development as a language was extremely quick with frequent feedback from engineers working on telephone switches in Erlang itself. These interactions proved processes-based concurrency and asynchronous message passing to be a good way to model the problems they faced. Moreover, the telephony world already had a certain culture going towards concurrency before Erlang came to be. This was inherited from PLEX, a language created earlier at Ericsson, and AXE, a switch developed with it. Erlang followed this tendency and attempted to improve on previous tools available.

Erlang had a few requirements to satisfy before being considered good. The main ones were being able to scale up and support many thousands of users across many switches, and then to achieve high reliability—to the point of never stopping the code.

Scalability

I'll focus on the scaling first. Some properties were seen as necessary to achieve scalability. Because users would be represented as processes which only reacted upon certain events (i.e.: receiving a call, hanging up, etc.), an ideal system would support processes doing small computations, switching between them very quickly as events came through. To make it efficient, it made sense for processes to be started very quickly, to be destroyed very quickly and to be able to switch them really fast. Having them lightweight

was mandatory to achieve this. It was also mandatory because you didn't want to have things like process pools (a fixed amount of processes you split the work between.) Instead, it would be much easier to design programs that could use as many processes as they need.

Another important aspect of scalability is to be able to bypass your hardware's limitations. There are two ways to do this: make the hardware better, or add more hardware. The first option is useful up to a certain point, after which it becomes extremely expensive (i.e.: buying a super computer). The second option is usually cheaper and requires you to add more computers to do the job. This is where distribution can be useful to have as a part of your language.

Anyway, to get back to small processes, because telephony applications needed a lot of reliability, it was decided that the cleanest way to do things was to forbid processes from sharing memory. Shared memory could leave things in an inconsistent state after some crashes (especially on data shared across different nodes) and had some complications. Instead, processes should communicate by sending messages where all the data is copied. This would risk being slower, but safer.

Fault-tolerance

This leads us on the second type of requirements for Erlang: reliability. The first writers of Erlang always kept in mind that

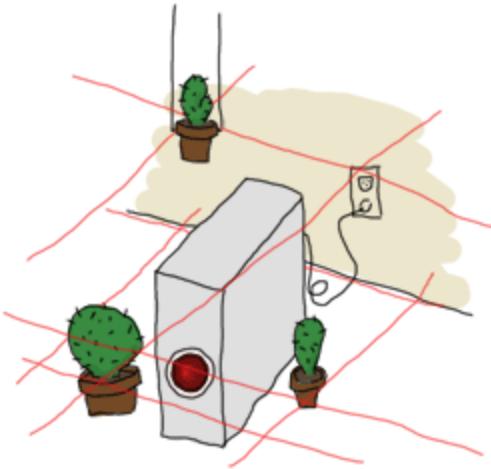
failure is common. You can try to prevent bugs all you want, but most of the time some of them will still happen. In the eventuality bugs don't happen, nothing can stop hardware failures all the time. The idea is thus to find good ways to handle errors and problems rather than trying to prevent them all.

It turns out that taking the design approach of multiple processes with message passing was a good idea, because error handling could be grafted onto it relatively easily. Take lightweight processes (made for quick restarts and shutdowns) as an example. Some studies proved that the main sources of downtime in large scale software systems are intermittent or transient bugs (source). Then, there's a principle that says that errors which corrupt data should cause the faulty part of the system to die as fast as possible in order to avoid propagating errors and bad data to the rest of the system. Another concept here is that there exist many different ways for a system to terminate, two of which are clean shutdowns and crashes (terminating with an unexpected error).

Here the worst case is obviously the crash. A safe solution would be to make sure all crashes are the same as clean shutdowns: this can be done through practices such as shared-nothing and single assignment (which isolates a process' memory), avoiding locks (a lock could happen to not be unlocked during a crash, keeping other processes from accessing the data or leaving data in an inconsistent

state) and other stuff I won't cover more, but were all part of Erlang's design. Your ideal solution in Erlang is thus to kill processes as fast as possible to avoid data corruption and transient bugs. Lightweight processes are a key element in this. Further error handling mechanisms are also part of the language to allow processes to monitor other processes (which are described in the [Errors and Processes](#) chapter), in order to know when processes die and to decide what to do about it.

Supposing restarting processes real fast is enough to deal with crashes, the next problem you get is hardware failures. How do you make sure your program keeps running when someone kicks the computer it's running on? Although a fancy defense mechanism comprising laser detection and strategically placed cacti could do the job for a while, it would not last forever. The hint is simply to have your program running on more than one computer at once, something that was needed for scaling anyway. This is another advantage of independent processes with no communication channel outside message passing. You can have them working the same way whether they're local or on a different computer, making fault tolerance through distribution nearly transparent to the programmer.



Being distributed has direct consequences on how processes can communicate with each other. One of the biggest hurdles of distribution is that you can't assume that because a node (a remote computer) was there when you made a function call, it will still be there for the whole transmission of the call or that it will even execute it correctly. Someone tripping over a cable or unplugging the machine would leave your application hanging. Or maybe it would make it crash. Who knows?

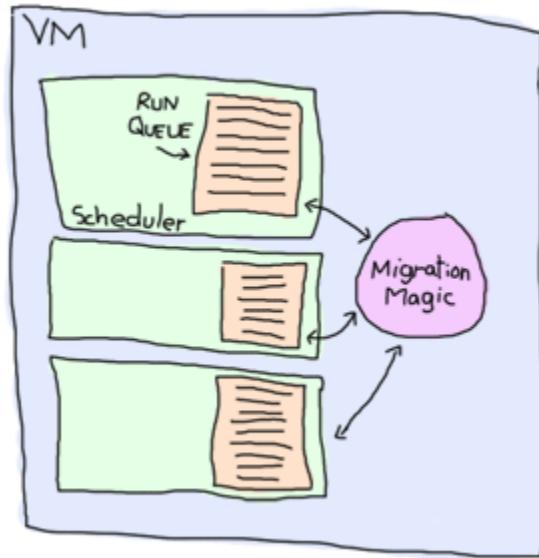
Well it turns out the choice of asynchronous message passing was a good design pick there too. Under the processes-with-asynchronous-messages model, messages are sent from one process to a second one and stored in a *mailbox* inside the receiving process until they are taken out to be read. It's important to mention that messages are sent without even checking if the receiving process exists or not because it would not be useful to do so. As implied in the previous paragraph, it's impossible to know if a process will crash between the time a message is sent and received. And

if it's received, it's impossible to know if it will be acted upon or again if the receiving process will die before that.

Asynchronous messages allow safe remote function calls because there is no assumption about what will happen; the programmer is the one to know. If you need to have a confirmation of delivery, you have to send a second message as a reply to the original process. This message will have the same safe semantics, and so will any program or library you build on this principle.

Implementation

Alright, so it was decided that lightweight processes with asynchronous message passing were the approach to take for Erlang. How to make this work? Well, first of all, the operating system can't be trusted to handle the processes. Operating systems have many different ways to handle processes, and their performance varies a lot. Most if not all of them are too slow or too heavy for what is needed by standard Erlang applications. By doing this in the VM, the Erlang implementers keep control of optimization and reliability. Nowadays, Erlang's processes take about 300 words of memory each and can be created in a matter of microseconds—not something doable on major operating systems these days.



To handle all these potential processes your programs could create, the VM starts one thread per core which acts as a *scheduler*. Each of these schedulers has a *run queue*, or a list of Erlang processes on which to spend a slice of time. When one of the schedulers has too many tasks in its run queue, some are migrated to another one. This is to say each Erlang VM takes care of doing all the load-balancing and the programmer doesn't need to worry about it. There are some other optimizations that are done, such as limiting the rate at which messages can be sent on overloaded processes in order to regulate and distribute the load.

All the hard stuff is in there, managed for you. That is what makes it easy to go parallel with Erlang. Going parallel means your program should go twice as fast if you add a second core, four times faster if there are 4 more and so on, right? It depends. Such a phenomenon is named *linear scaling* in relation to speed gain vs. the number of cores or

processors (see the graph below.) In real life, there is no such thing as a free lunch (well, there are at funerals, but someone still has to pay, somewhere).

Not Entirely Unlike Linear Scaling

The difficulty of obtaining linear scaling is not due to the language itself, but rather to the nature of the problems to solve. Problems that scale very well are often said to be *embarrassingly parallel*. If you look for embarrassingly parallel problems on the Internet, you're likely to find examples such as ray-tracing (a method to create 3D images), brute-forcing searches in cryptography, weather prediction, etc.

From time to time, people then pop up in IRC channels, forums or mailing lists asking if Erlang could be used to solve that kind of problem, or if it could be used to program on a GPU. The answer is almost always 'no'. The reason is relatively simple: all these problems are usually about numerical algorithms with lots of data crunching. Erlang is not very good at this.

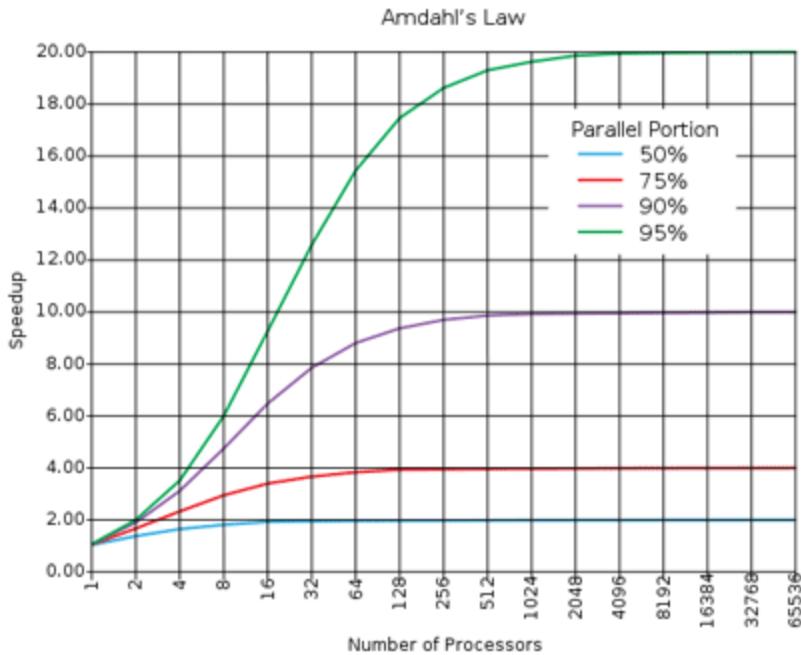
Erlang's embarrassingly parallel problems are present at a higher level. Usually, they have to do with concepts such as chat servers, phone switches, web servers, message queues, web crawlers or any other application where the work done can be represented as independent logical entities (actors, anyone?). This kind of problem can be solved efficiently with close-to-linear scaling.

Many problems will never show such scaling properties. In fact, you only need one centralized sequence of operations to lose it all. **Your parallel program only goes as fast as its slowest sequential part.** An example of that phenomenon is observable any time you go to a mall. Hundreds of people can be shopping at once, rarely interfering with each other. Then once it's time to pay, queues form as soon as there are fewer cashiers than there are customers ready to leave.

It would be possible to add cashiers until there's one for each customer, but then you would need a door for each customer because they couldn't get inside or outside the mall all at once.

To put this another way, even though customers could pick each of their items in parallel and basically take as much time to shop whether they're alone or a thousand in the store, they would still have to wait to pay. Therefore their shopping experience can never be shorter than the time it takes them to wait in the queue and pay.

A generalisation of this principle is called Amdahl's Law. It indicates how much of a speedup you can expect your system to have whenever you add parallelism to it, and in what proportion:



According to Amdahl's law, code that is 50% parallel can never get faster than twice what it was before, and code that is 95% parallel can theoretically be expected to be about 20 times faster if you add enough processors. What's interesting to see on this graph is how getting rid of the last few sequential parts of a program allows a relatively huge theoretical speedup compared to removing as much sequential code in a program that is not very parallel to begin with.

Don't drink too much Kool-Aid:

Parallelism is *not* the answer to every problem. In some cases, going parallel will even slow down your application. This can happen whenever your program is 100% sequential, but still uses multiple processes.

One of the best examples of this is the *ring benchmark*. A ring benchmark is a test where many thousands of processes will pass a piece of data to one after the other in a circular manner. Think of it as a game of telephone if you want. In this benchmark, only one process at a time does something useful, but the Erlang VM still spends time distributing the load across cores and giving every process its share of time.

This plays against many common hardware optimizations and makes the VM spend time doing useless stuff. This often makes purely sequential applications run much slower on many cores than on a single one. In this case, disabling symmetric multiprocessing (`$ erl -smp disable`) might be a good idea.

So long and thanks for all the fish!

Of course, this chapter would not be complete if it wouldn't show the three primitives required for concurrency in Erlang: spawning new processes, sending messages, and receiving messages. In practice there are more mechanisms required for making really reliable applications, but for now this will suffice.

I've skipped around the issue a whole lot and I have yet to explain what a process really is. It's in fact nothing but a function. That's it. It runs a function and once it's done, it disappears. Technically, a process also has some hidden

state (such as a mailbox for messages), but functions are enough for now.

To start a new process, Erlang provides the function `spawn/1`, which takes a single function and runs it:

```
1> F = fun() -> 2 + 2 end.  
#Fun<erl_eval.20.67289768>  
2> spawn(F).  
<0.44.0>
```

The result of `spawn/1` (`<0.44.0>`) is called a *Process Identifier*, often just written *PID*, *Pid*, or *pid* by the community. The process identifier is an arbitrary value representing any process that exists (or might have existed) at some point in the VM's life. It is used as an address to communicate with the process.

You'll notice that we can't see the result of the function *F*. We only get its pid. That's because processes do not return anything.

How can we see the result of *F* then? Well, there are two ways. The easiest one is to just output whatever we get:

```
3> spawn(fun() -> io:format("~p~n",[2 + 2]) end).  
4  
<0.46.0>
```

This isn't practical for a real program, but it is useful for seeing how Erlang dispatches processes. Fortunately, using `io:format/2` is enough to let us experiment. We'll start 10

processes real quick and pause each of them for a while with the help of the function `timer:sleep/1`, which takes an integer value N and waits for N milliseconds before resuming code. After the delay, the value present in the process is output.

```
4> G = fun(X) -> timer:sleep(10), io:format("~p~n", [X]) end.  
#Fun<erl_eval.6.13229925>  
5> [spawn(fun() -> G(X) end) || X <- lists:seq(1,10)].  
[<0.273.0>,<0.274.0>,<0.275.0>,<0.276.0>,<0.277.0>,  
<0.278.0>,<0.279.0>,<0.280.0>,<0.281.0>,<0.282.0>]  
2  
1  
4  
3  
5  
8  
7  
6  
10  
9
```

The order doesn't make sense. Welcome to parallelism. Because the processes are running at the same time, the ordering of events isn't guaranteed anymore. That's because the Erlang VM uses many tricks to decide when to run a process or another one, making sure each gets a good share of time. Many Erlang services are implemented as processes, including the shell you're typing in. Your processes must be balanced with those the system itself needs and this might be the cause of the weird ordering.

Note: the results are similar whether symmetric multiprocessing is enabled or not. To prove it, you can just test it out by starting the Erlang VM with \$ erl -smp disable.

To see if your Erlang VM runs with or without SMP support in the first place, start a new VM without any options and look for the first line output. If you can spot the text [smp:2:2] [rq:2], it means you're running with SMP enabled, and that you have 2 run queues (rq, or schedulers) running on two cores. If you only see [rq:1], it means you're running with SMP disabled.

If you wanted to know, [smp:2:2] means there are two cores available, with two schedulers. [rq:2] means there are two run queues active. In earlier versions of Erlang, you could have multiple schedulers, but with only one shared run queue. Since R13B, there is one run queue per scheduler by default; this allows for better parallelism.

To prove the shell itself is implemented as a regular process, I'll use the BIF self/0, which returns the pid of the current process:

```
6> self().  
<0.41.0>  
7> exit(self()).  
** exception exit: <0.41.0>  
8> self().  
<0.285.0>
```

And the pid changes because the process has been restarted. The details of how this works will be seen later. For

now, there's more basic stuff to cover. The most important one right now is to figure out how to send messages around, because nobody wants to be stuck with outputting the resulting values of processes all the time, and then entering them by hand in other processes (at least I know I don't.)

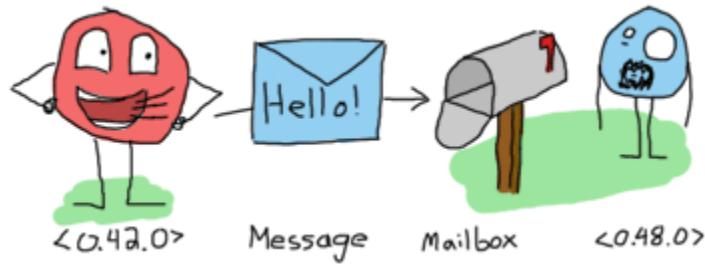
The next primitive required to do message passing is the operator !, also known as the *bang* symbol. On the left-hand side it takes a pid and on the right-hand side it takes any Erlang term. The term is then sent to the process represented by the pid, which can access it:

```
9> self() ! hello.  
hello
```

The message has been put in the process' mailbox, but it hasn't been read yet. The second hello shown here is the return value of the send operation. This means it is possible to send the same message to many processes by doing:

```
10> self() ! self() ! double.  
double
```

Which is equivalent to `self() ! (self() ! double)`. A thing to note about a process' mailbox is that the messages are kept in the order they are received. Every time a message is read it is taken out of the mailbox. Again, this is a bit similar to the introduction's example with people writing letters.



To see the contents of the current mailbox, you can use the `flush()` command while in the shell:

```
11> flush().
Shell got hello
Shell got double
Shell got double
ok
```

This function is just a shortcut that outputs received messages. This means we still can't bind the result of a process to a variable, but at least we know how to send it from a process to another one and check if it's been received.

Sending messages that nobody will read is as useful as writing emo poetry; not a whole lot. This is why we need the `receive` statement. Rather than playing for too long in the shell, we'll write a short program about dolphins to learn about it:

```
-module(dolphins).
-compile(export_all).

dolphin1() ->
    receive
        do_a_flip ->
```

```

io:format("How about no?~n");
fish ->
    io:format("So long and thanks for all the fish!~n");
_ ->
    io:format("Heh, we're smarter than you humans.~n")
end.
```

As you can see, receive is syntactically similar to case ... of. In fact, the patterns work exactly the same way except they bind variables coming from messages rather than the expression between case and of. Receives can also have guards:

```

receive
    Pattern1 when Guard1 -> Expr1;
    Pattern2 when Guard2 -> Expr2;
    Pattern3 -> Expr3
end
```

We can now compile the above module, run it, and start communicating with dolphins:

```

11> c(dolphins).
{ok,dolphins}
12> Dolphin = spawn(dolphins, dolphin1, []).
<0.40.0>
13> Dolphin ! "oh, hello dolphin!".
Heh, we're smarter than you humans.
"oh, hello dolphin!"
14> Dolphin ! fish.
fish
15>
```

Here we introduce a new way of spawning with `spawn/3`. Rather than taking a single function, `spawn/3` takes the module, function and its arguments as its own arguments. Once the function is running, the following events take place:

1. The function hits the `receive` statement. Given the process' mailbox is empty, our dolphin waits until it gets a message;
2. The message "oh, hello dolphin!" is received. The function tries to pattern match against `do_a_flip`. This fails, and so the pattern `fish` is tried and also fails. Finally, the message meets the catch-all clause `(_)` and matches.
3. The process outputs the message "Heh, we're smarter than you humans."

Then it should be noted that if the first message we sent worked, the second provoked no reaction whatsoever from the process `<0.40.0>`. This is due to the fact once our function output "Heh, we're smarter than you humans.", it terminated and so did the process. We'll need to restart the dolphin:

```
8> f(Dolphin).  
ok  
9> Dolphin = spawn(dolphins, dolphin1, []).  
<0.53.0>  
10> Dolphin ! fish.  
So long and thanks for all the fish!  
fish
```

And this time the `fish` message works. Wouldn't it be useful to be able to receive a reply from the dolphin rather than

having to use `io:format/2`? Of course it would (why am I even asking?) I've mentioned earlier in this chapter that the only manner to know if a process had received a message is to send a reply. Our dolphin process will need to know who to reply to. This works like it does with the postal service. If we want someone to know answer our letter, we need to add our address. In Erlang terms, this is done by packaging a process' pid in a tuple. The end result is a message that looks a bit like `{Pid, Message}`. Let's create a new dolphin function that will accept such messages:

```
dolphin2() ->
    receive
        {From, do_a_flip} ->
            From ! "How about no?";
        {From, fish} ->
            From ! "So long and thanks for all the fish!";
        _ ->
            io:format("Heh, we're smarter than you humans.\n")
    end.
```

As you can see, rather than accepting `do_a_flip` and `fish` for messages, we now require a variable `From`. That's where the process identifier will go.

```
11> c(dolphins).
{ok,dolphins}
12> Dolphin2 = spawn(dolphins, dolphin2, []).
<0.65.0>
13> Dolphin2 ! {self(), do_a_flip}.
{<0.32.0>,do_a_flip}
14> flush().
```

Shell got "How about no?"

ok

It seems to work pretty well. We can receive replies to messages we sent (we need to add an address to each message), but we still need to start a new process for each call. Recursion is the way to solve this problem. We just need the function to call itself so it never ends and always expects more messages. Here's a function dolphin3/0 that puts this in practice:

```
dolphin3() ->
    receive
        {From, do_a_flip} ->
            From ! "How about no?",
            dolphin3();
        {From, fish} ->
            From ! "So long and thanks for all the fish!";
        _ ->
            io:format("Heh, we're smarter than you humans.~n"),
            dolphin3()
    end.
```

Here the catch-all clause and the do_a_flip clause both loop with the help of dolphin3/0. Note that the function will not blow the stack because it is tail recursive. As long as only these messages are sent, the dolphin process will loop indefinitely. However, if we send the fish message, the process will stop:

```
15> Dolphin3 = spawn(dolphins, dolphin3, []).
<0.75.0>
16> Dolphin3 ! Dolphin3 ! {self(), do_a_flip}.
```

```
{<0.32.0>,do_a_flip}  
17> flush().  
Shell got "How about no?"  
Shell got "How about no?"  
ok  
18> Dolphin3 ! {self(), unknown_message}.  
Heh, we're smarter than you humans.  
{<0.32.0>,unknown_message}  
19> Dolphin3 ! Dolphin3 ! {self(), fish}.  
{<0.32.0>,fish}  
20> flush().  
Shell got "So long and thanks for all the fish!"  
ok
```

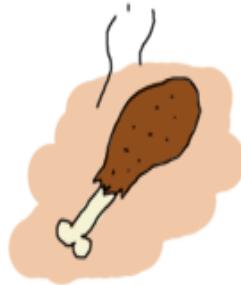
And that should be it for dolphins.erl. As you see, it does respect our expected behavior of replying once for every message and keep going afterwards, except for the fish call. The dolphin got fed up with our crazy human antics and left us for good.



There you have it. This is the core of all of Erlang's concurrency. We've seen processes and basic message passing. There are more concepts to see in order to make truly useful and reliable programs. We'll see some of them in the next chapter, and more in the chapters after that.

More On Multiprocessing

State Your State



The examples shown in the previous chapter were all right for demonstrative purposes, but you won't go far with only that in your toolkit. It's not that the examples were bad, it's mostly that there is not a huge advantage to processes and actors if they're just functions with messages. To fix this, we have to be able to hold state in a process.

Let's first create a function in a new `kitchen.erl` module that will let a process act like a fridge. The process will allow two operations: storing food in the fridge and taking food from the fridge. It should only be possible to take food that has been stored beforehand. The following function can act as the base for our process:

```
-module(kitchen).  
-compile(export_all).
```

```
fridge1() ->  
    receive
```

```

{From, {store, _Food}} ->
    From ! {self(), ok},
    fridge1();
{From, {take, _Food}} ->
    %% uh....
    From ! {self(), not_found},
    fridge1();
terminate ->
    ok
end.

```

Something's wrong with it. When we ask to store the food, the process should reply with `ok`, but there is nothing actually storing the food; `fridge1()` is called and then the function starts from scratch, without state. You can also see that when we call the process to take food from the fridge, there is no state to take it from and so the only thing to reply is `not_found`. In order to store and take food items, we'll need to add state to the function.

With the help of recursion, the state to a process can then be held entirely in the parameters of the function. In the case of our fridge process, a possibility would be to store all the food as a list, and then look in that list when someone needs to eat something:

```

fridge2(FoodList) ->
    receive
        {From, {store, Food}} ->
            From ! {self(), ok},
            fridge2([Food|FoodList]);
        {From, {take, Food}} ->

```

```

case lists:member(Food, FoodList) of
    true ->
        From ! {self(), {ok, Food}},
        fridge2(lists:delete(Food, FoodList));
    false ->
        From ! {self(), not_found},
        fridge2(FoodList)
end;
terminate ->
    ok
end.

```

The first thing to notice is that `fridge2/1` takes one argument, `FoodList`. You can see that when we send a message that matches `{From, {store, Food}}`, the function will add `Food` to `FoodList` before going. Once that recursive call is made, it will then be possible to retrieve the same item. In fact, I implemented it there. The function uses `lists:member/2` to check whether `Food` is part of `FoodList` or not. Depending on the result, the item is sent back to the calling process (and removed from `FoodList`) or `not_found` is sent back otherwise:

```

1> c(kitchen).
{ok,kitchen}
2> Pid = spawn(kitchen, fridge2, [[baking_soda]]).
<0.51.0>
3> Pid ! {self(), {store, milk}}.
{<0.33.0>,{store,milk}}
4> flush().
Shell got {<0.51.0>,ok}
ok

```

Storing items in the fridge seems to work. We'll try with some more stuff and then try to take it from the fridge.

```
5> Pid ! {self(), {store, bacon}}.  
{<0.33.0>,{store,bacon}}  
6> Pid ! {self(), {take, bacon}}.  
{<0.33.0>,{take,bacon}}  
7> Pid ! {self(), {take, turkey}}.  
{<0.33.0>,{take,turkey}}  
8> flush().  
Shell got {<0.51.0>,ok}  
Shell got {<0.51.0>,{ok,bacon}}  
Shell got {<0.51.0>,not_found}  
ok
```

As expected, we can take bacon from the fridge because we have put it in there first (along with the milk and baking soda), but the fridge process has no turkey to find when we request some. This is why we get the last {<0.51.0>,not_found} message.

We love messages, but we keep them secret

Something annoying with the previous example is that the programmer who's going to use the fridge has to know about the protocol that's been invented for that process. That's a useless burden. A good way to solve this is to abstract messages away with the help of functions dealing with receiving and sending them:

```
store(Pid, Food) ->  
  Pid ! {self(), {store, Food}},
```

```
receive
  {Pid, Msg} -> Msg
end.
```

```
take(Pid, Food) ->
  Pid ! {self(), {take, Food}},
receive
  {Pid, Msg} -> Msg
end.
```

Now the interaction with the process is much cleaner:

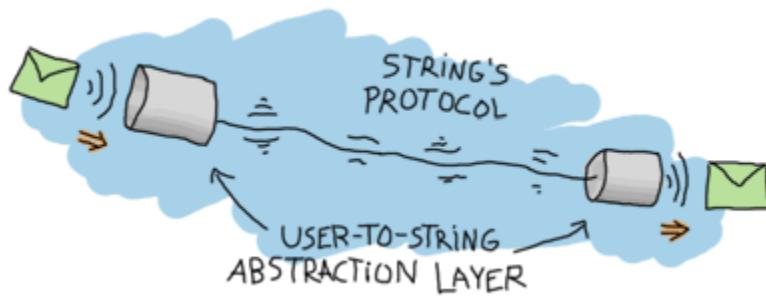
```
9> c(kitchen).
{ok,kitchen}
10> f().
ok
11> Pid = spawn(kitchen, fridge2, [[baking_soda]]).
<0.73.0>
12> kitchen:store(Pid, water).
ok
13> kitchen:take(Pid, water).
{ok,water}
14> kitchen:take(Pid, juice).
not_found
```

We don't have to care about how the messages work anymore, if sending `self()` or a precise atom like `take` or `store` is needed: all that's needed is a pid and knowing what functions to call. This hides all of the dirty work and makes it easier to build on the fridge process.

One thing left to do would be to hide that whole part about needing to spawn a process. We dealt with hiding messages,

but then we still expect the user to handle the creation of the process. I'll add the following start/1 function:

```
start(FoodList) ->  
    spawn(?MODULE, fridge2, [FoodList]).
```



Here, ?MODULE is a macro returning the current module's name. It doesn't look like there are any advantages to writing such a function, but there really are some. The essential part of it would be consistency with the calls to take/2 and store/2: everything about the fridge process is now handled by the kitchen module. If you were to add logging when the fridge process is started or start a second process (say a freezer), it would be really easy to do inside our start/1 function. However if the spawning is left for the user to do through spawn/3, then every place that starts a fridge now needs to add the new calls. That's prone to errors and errors suck.

Let's see this function put to use:

```
15> f().  
ok  
16> c(kitchen).  
{ok,kitchen}  
17> Pid = kitchen:start([rhubarb, dog, hotdog]).
```

```
<0.84.0>
18> kitchen:take(Pid, dog).
{ok,dog}
19> kitchen:take(Pid, dog).
not_found
```

Yay! The dog has got out of the fridge and our abstraction is complete!

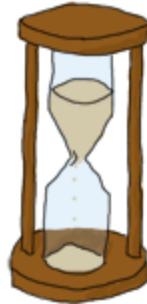
Time Out

Let's try a little something with the help of the command `pid(A,B,C)`, which lets us change the 3 integers A , B and C into a pid. Here we'll deliberately feed `kitchen:take/2` a fake one:

```
20> kitchen:take(pid(0,250,0), dog).
```

Woops. The shell is frozen. This happened because of how `take/2` was implemented. To understand what goes on, let's first revise what happens in the normal case:

1. A message to take food is sent from you (the shell) to the fridge process;
2. Your process switches to receive mode and waits for a new message;
3. The fridge removes the item and sends it to your process;
4. Your process receives it and moves on with its life.



And here's what happens when the shell freezes:

1. A message to take food is sent from you (the shell) to an unknown process;
2. Your process switches to receive mode and waits for a new message;
3. The unknown process either doesn't exist or doesn't expect such a message and does nothing with it;
4. Your shell process is stuck in receive mode.

That's annoying, especially because there is no error handling possible here. Nothing illegal happened, the program is just waiting. In general, anything dealing with asynchronous operations (which is how message passing is done in Erlang) needs a way to give up after a certain period of time if it gets no sign of receiving data. A web browser does it when a page or image takes too long to load, you do it when someone takes too long before answering the phone or is late at a meeting. Erlang certainly has an appropriate mechanism for that, and it's part of the receive construct:

receive

 Match -> Expression1

 after Delay ->

```
Expression2
end.
```

The part in between `receive` and `after` is exactly the same that we already know. The `after` part will be triggered if as much time as `Delay` (an integer representing milliseconds) has been spent without receiving a message that matches the `Match` pattern. When this happens, `Expression2` is executed.

We'll write two new interface functions, `store2/2` and `take2/2`, which will act exactly like `store/2` and `take/2` with the exception that they will stop waiting after 3 seconds:

```
store2(Pid, Food) ->
  Pid ! {self(), {store, Food}},
  receive
    {Pid, Msg} -> Msg
  after 3000 ->
    timeout
  end.
```

```
take2(Pid, Food) ->
  Pid ! {self(), {take, Food}},
  receive
    {Pid, Msg} -> Msg
  after 3000 ->
    timeout
  end.
```

Now you can unfreeze the shell with [^G](#) and try the new interface functions:

User switch command

--> k

--> s

--> c

Eshell V5.7.5 (abort with ^G)

1> c(kitchen).

{ok,kitchen}

2> kitchen:take2(pid(0,250,0), dog).

timeout

And now it works.

Note: I said that `after` only takes milliseconds as a value, but it is actually possible to use the atom infinity. While this is not useful in many cases (you might just remove the `after` clause altogether), it is sometimes used when the programmer can submit the wait time to a function where receiving a result is expected. That way, if the programmer really wants to wait forever, he can.

There are uses to such timers other than giving up after too long. One very simple example is how the `timer:sleep/1` function we've used before works. Here's how it is implemented (let's put it in a new `multiproc.erl` module):

```
sleep(T) ->
    receive
        after T -> ok
    end.
```

In this specific case, no message will ever be matched in the `receive` part of the construct because there is no pattern.

Instead, the after part of the construct will be called once the delay T has passed.

Another special case is when the timeout is at 0:

```
flush() ->
    receive
        _ -> flush()
    after 0 ->
        ok
    end.
```

When that happens, the Erlang VM will try and find a message that fits one of the available patterns. In the case above, anything matches. As long as there are messages, the `flush/0` function will recursively call itself until the mailbox is empty. Once this is done, the `after 0 -> ok` part of the code is executed and the function returns.

Selective Receives

This 'flushing' concept makes it possible to implement a *selective receive* which can give a priority to the messages you receive by nesting calls:

```
important() ->
    receive
        {Priority, Message} when Priority > 10 ->
            [Message | important()]
    after 0 ->
        normal()
    end.
```

```
normal() ->
    receive
        {_, Message} ->
            [Message | normal()]
    after 0 ->
        []
end.
```

This function will build a list of all messages with those with a priority above 10 coming first:

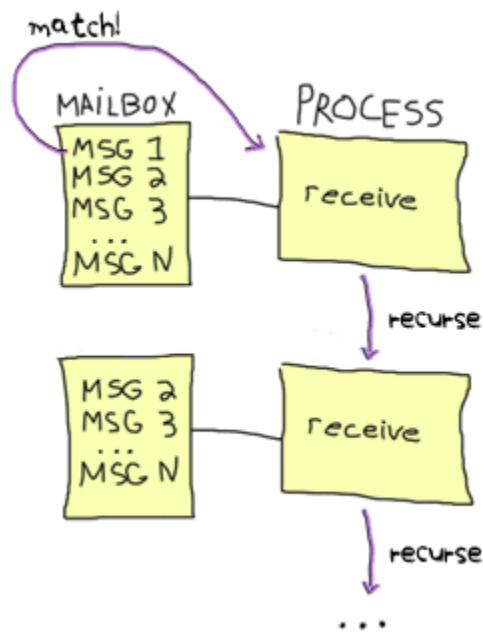
```
1> c(multiproc).
{ok,multiproc}
2> self() ! {15, high}, self() ! {7, low}, self() ! {1, low}, self() ! {17, high}.
{17,high}
3> multiproc:important().
[high,high,low,low]
```

Because I used the `after 0` bit, every message will be obtained until none is left, but the process will try to grab all those with a priority above 10 before even considering the other messages, which are accumulated in the `normal/0` call.

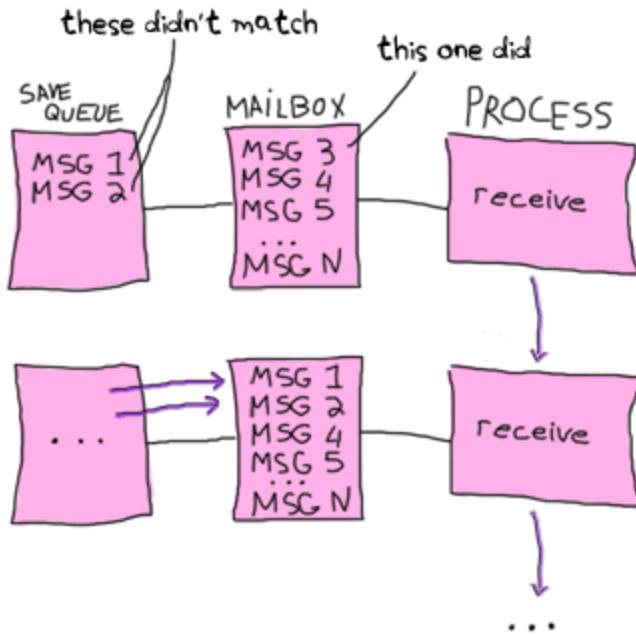
If this practice looks interesting, be aware that it is sometimes unsafe due to the way selective receives work in Erlang.

When messages are sent to a process, they're stored in the mailbox until the process reads them and they match a pattern there. As said in the [previous chapter](#), the messages are stored in the order they were received. This means every time you match a message, it begins by the oldest one.

That oldest message is then tried against every pattern of the receive until one of them matches. When it does, the message is removed from the mailbox and the code for the process executes normally until the next receive. When this next receive is evaluated, the VM will look for the oldest message currently in the mailbox (the one after the one we removed), and so on.



When there is no way to match a given message, it is put in a save queue and the next message is tried. If the second message matches, the first message is put back on top of the mailbox to be retried later.



This lets you only care about the messages that are useful. Ignoring some messages to handle them later in the manner described above is the essence of *selective receives*. While they're useful, the problem with them is that if your process has a lot of messages you never care about, reading useful messages will actually take longer and longer (and the processes will grow in size too).

In the drawing above, imagine we want the 367th message, but the first 366 are junk ignored by our code. To get the 367th message, the process needs to try to match the 366 first ones. Once it's done and they've all been put in the queue, the 367th message is taken out and the first 366 are put back on top of the mailbox. The next useful message could be burrowed much deeper and take even longer to be found.

This kind of receive is a frequent cause of performance problems in Erlang. If your application is running slow and you know there are lots of messages going around, this could be the cause.

If such selective receives are effectively causing a massive slowdown in your code, the first thing to do is to ask yourself is why you are getting messages you do not want. Are the messages sent to the right processes? Are the patterns correct? Are the messages formatted incorrectly? Are you using one process where there should be many? Answering one or many of these questions could solve your problem.

Because of the risks of having useless messages polluting a process' mailbox, Erlang programmers sometimes take a defensive measure against such events. A standard way to do it might look like this:

```
receive
    Pattern1 -> Expression1;
    Pattern2 -> Expression2;
    Pattern3 -> Expression3;
    ...
    PatternN -> ExpressionN;
    Unexpected ->
        io:format("unexpected message ~p~n", [Unexpected])
end.
```

What this does is make sure any message will match at least one clause. The *Unexpected* variable will match anything, take the unexpected message out of the mailbox and show

a warning. Depending on your application, you might want to store the message into some kind of logging facility where you will be able to find information about it later on: if the messages are going to the wrong process, it'd be a shame to lose them for good and have a hard time finding why that other process doesn't receive what it should.

In the case you do need to work with a priority in your messages and can't use such a catch-all clause, a smarter way to do it would be to implement a min-heap or use the `gb_trees` module and dump every received message in it (make sure to put the priority number first in the key so it gets used for sorting the messages). Then you can just search for the smallest or largest element in the data structure according to your needs.

In most cases, this technique should let you receive messages with a priority more efficiently than selective receives. However, it could slow you down if most messages you receive have the highest priority possible. As usual, the trick is to profile and measure before optimizing.

Note: Since R14A, a new optimization has been added to Erlang's compiler. It simplifies selective receives in very specific cases of back-and-forth communications between processes. An example of such a function is `optimized/1` in `multiproc.erl`.

To make it work, a reference (`make_ref()`) has to be created in a function and then sent in a message. In the same function,

a selective receive is then made. If no message can match unless it contains the same reference, the compiler automatically makes sure the VM will skip messages received before the creation of that reference.

Note that you shouldn't try to coerce your code to fit such optimizations. The Erlang developers only look for patterns that are frequently used and then make them faster. If you write idiomatic code, optimizations should come to you. Not the other way around.

With these concepts understood, the next step will be to do error handling with multiple processes.

Errors and Processes

Links

A link is a specific kind of relationship that can be created between two processes. When that relationship is set up and one of the processes dies from an unexpected throw, error or exit (see [Errors and Exceptions](#)), the other linked process also dies.

This is a useful concept from the perspective of failing as soon as possible to stop errors: if the process that has an error crashes but those that depend on it don't, then all these depending processes now have to deal with a dependency disappearing. Letting them die and then restarting the whole group is usually an acceptable alternative. Links let us do exactly this.

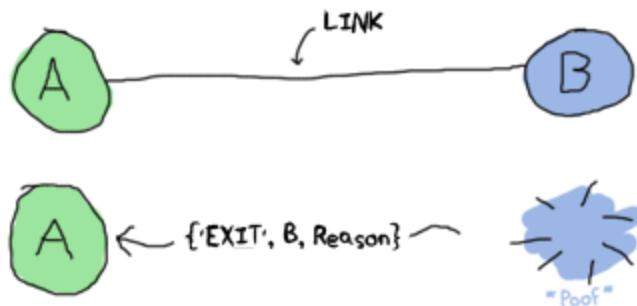
To set a link between two processes, Erlang has the primitive function `link/1`, which takes a `Pid` as an argument. When called, the function will create a link between the current process and the one identified by `Pid`. To get rid of a link, use `unlink/1`. When one of the linked processes crashes, a special kind of message is sent, with information relative to what happened. No such message is sent if the process dies of natural causes (read: is done running its functions.) I'll first introduce this new function as part of `linkmon.erl`:

```
myproc() ->  
    timer:sleep(5000),  
    exit(reason).
```

If you try the next following calls (and wait 5 seconds between each spawn command), you should see the shell crashing for 'reason' only when a link has been set between the two processes.

```
1> c(linkmon).  
{ok,linkmon}  
2> spawn(fun linkmon:myproc/0).  
<0.52.0>  
3> link(spawn(fun linkmon:myproc/0)).  
true  
** exception error: reason
```

Or, to put it in picture:



However, this '{EXIT, B, Reason}' message can not be caught with a try ... catch as usual. Other mechanisms need to be used to do this. We'll see them later.

It's important to note that links are used to establish larger groups of processes that should all die together:

```

chain(0) ->
    receive
        _ -> ok
    after 2000 ->
        exit("chain dies here")
    end;
chain(N) ->
    Pid = spawn(fun() -> chain(N-1) end),
    link(Pid),
    receive
        _ -> ok
    end.

```

This function will take an integer N , start N processes linked one to the other. In order to be able to pass the $N-1$ argument to the next 'chain' process (which calls `spawn/1`), I wrap the call inside an anonymous function so it doesn't need arguments anymore. Calling `spawn(?MODULE, chain, [N-1])` would have done a similar job.

Here, I'll have many processes linked together, dying as each of their successors exits:

```

4> c(linkmon).
{ok,linkmon}
5> link(spawn(linkmon, chain, [3])).
true
** exception error: "chain dies here"

```

And as you can see, the shell does receive the death signal from some other process. Here's a drawn representation of the spawned processes and links going down:

```
[shell] == [3] == [2] == [1] == [0]
[shell] == [3] == [2] == [1] == *dead*
[shell] == [3] == [2] == *dead*
[shell] == [3] == *dead*
[shell] == *dead*
*dead, error message shown*
[shell] <-- restarted
```

After the process running `linkmon:chain(0)` dies, the error is propagated down the chain of links until the shell process itself dies because of it. The crash could have happened in any of the linked processes; because links are bidirectional, you only need one of them to die for the others to follow suit.

Note: If you wanted to kill another process from the shell, you could use the function `exit/2`, which is called this way: `exit(Pid, Reason)`. Try it if you wish.

Note: Links can not be stacked. If you call `link/1` 15 times for the same two processes, only one link will still exist between them and a single call to `unlink/1` will be enough to tear it down.

It's important to note that `link(spawn(Function))` or `link(spawn(M,F,A))` happens in more than one step. In some cases, it is possible for a process to die before the link has been set up and then provoke unexpected behavior. For this reason, the function `spawn_link/1-3` has been added to the language. It takes the same arguments as `spawn/1-3`, creates a process and links it as if `link/1` had been there, except it's all done as an atomic operation (the operations are combined

as a single one, which can either fail or succeed, but nothing else). This is generally considered safer and you save a set of parentheses too.



It's a Trap!

Now to get back to links and processes dying. Error propagation across processes is done through a process similar to message passing, but with a special type of message called signals. Exit signals are 'secret' messages that automatically act on processes, killing them in the action.

I have mentioned many times already that in order to be reliable, an application needs to be able to both kill and restart a process quickly. Right now, links are alright to do the killing part. What's missing is the restarting.

In order to restart a process, we need a way to first know that it died. This can be done by adding a layer on top of links (the delicious frosting on the cake) with a concept called *system processes*. System processes are basically normal processes, except they can convert exit signals to regular messages. This is done by calling `process_flag(trap_exit, true)` in a running process. Nothing speaks as much as an example, so we'll go with that. I'll just redo the chain example with a system process at the beginning:

```
1> process_flag(trap_exit, true).
true
2> spawn_link(fun() -> linkmon:chain(3) end).
<0.49.0>
3> receive X -> X end.
{'EXIT',<0.49.0>,"chain dies here"}
```

Ah! Now things get interesting. To get back to our drawings, what happens is now more like this:

```
[shell] == [3] == [2] == [1] == [0]
[shell] == [3] == [2] == [1] == *dead*
[shell] == [3] == [2] == *dead*
[shell] == [3] == *dead*
[shell] <-- {'EXIT,Pid,"chain dies here"} -- *dead*
[shell] <-- still alive!
```

And this is the mechanism allowing for a quick restart of processes. By writing programs using system processes, it is easy to create a process whose only role is to check if something dies and then restart it whenever it fails. We'll

cover more of this in the next chapter, when we really apply these techniques.

For now, I want to come back to the exception functions seen in the [exceptions chapter](#) and show how they behave around processes that trap exits. Let's first set the bases to experiment without a system process. I'll successively show the results of uncaught throws, errors and exits in neighboring processes:

Exception source: spawn_link(fun() -> ok end)

Untrapped Result: - nothing -

Trapped Result: {'EXIT', <0.61.0>, normal}

The process exited normally, without a problem. Note that this looks a bit like the result of catch exit(normal), except a PID is added to the tuple to know what process failed.

Exception source: spawn_link(fun() -> exit(reason) end)

Untrapped Result: ** exception exit: reason

Trapped Result: {'EXIT', <0.55.0>, reason}

The process has terminated for a custom reason. In this case, if there is no trapped exit, the process crashes. Otherwise, you get the above message.

Exception source: spawn_link(fun() -> exit(normal) end)

Untrapped Result: - nothing -

Trapped Result: {'EXIT', <0.58.0>, normal}

This successfully emulates a process terminating normally. In some cases, you might want to kill a process

as part of the normal flow of a program, without anything exceptional going on. This is the way to do it.

Exception source: `spawn_link(fun() -> 1/0 end)`

Untrapped Result: Error in process <0.44.0> with exit value:

`{badarith, [{erlang, '/', [1,0]}]}`

Trapped Result: `{'EXIT', <0.52.0>, {badarith, [{erlang, '/', [1,0]}]}}`

The error (`{badarith, Reason}`) is never caught by a try ... catch block and bubbles up into an 'EXIT'. At this point, it behaves exactly the same as `exit(reason)` did, but with a stack trace giving more details about what happened.

Exception source: `spawn_link(fun() -> erlang:error(reason) end)`

Untrapped Result: Error in process <0.47.0> with exit value: `{reason, [{erlang, apply, 2}]}`

Trapped Result: `{'EXIT', <0.74.0>, {reason, [{erlang, apply, 2}]}}`

Pretty much the same as with 1/0. That's normal, `erlang:error/1` is meant to allow you to do just that.

Exception source: `spawn_link(fun() -> throw(rocks) end)`

Untrapped Result: Error in process <0.51.0> with exit value:

`{nocatch, rocks}, [{erlang, apply, 2}]}`

Trapped Result: `{'EXIT', <0.79.0>, {{nocatch, rocks}, [{erlang, apply, 2}]}}`

Because the throw is never caught by a try ... catch, it bubbles up into an error, which in turn bubbles up into an EXIT. Without trapping exit, the process fails. Otherwise it deals with it fine.

And that's about it for usual exceptions. Things are normal: everything goes fine. Exceptional stuff happens: processes die, different signals are sent around.

Then there's `exit/2`. This one is the Erlang process equivalent of a gun. It allows a process to kill another one from a distance, safely. Here are some of the possible calls:

Exception source: `exit(self(), normal)`

Untrapped Result: ** exception exit: normal

Trapped Result: {'EXIT', <0.31.0>, normal}

When not trapping exits, `exit(self(), normal)` acts the same as `exit(normal)`. Otherwise, you receive a message with the same format you would have had by listening to links from foreign processes dying.

Exception source: `exit(spawn_link(fun() -> timer:sleep(50000) end), normal)`

Untrapped Result: - nothing -

Trapped Result: - nothing -

This basically is a call to `exit(Pid, normal)`. This command doesn't do anything useful, because a process can not be remotely killed with the reason `normal` as an argument.

Exception source: `exit(spawn_link(fun() -> timer:sleep(50000) end), reason)`

Untrapped Result: ** exception exit: reason

Trapped Result: {'EXIT', <0.52.0>, reason}

This is the foreign process terminating for reason itself.

Looks the same as if the foreign process called `exit(reason)` on itself.

Exception source: `exit(spawn_link(fun() -> timer:sleep(50000) end), kill)`

Untrapped Result: ** exception exit: killed

Trapped Result: {'EXIT', <0.58.0>, killed}

Surprisingly, the message gets changed from the dying process to the spawner. The spawner now receives killed instead of kill. That's because kill is a special exit signal.

More details on this later.

Exception source: `exit(self(), kill)`

Untrapped Result: ** exception exit: killed

Trapped Result: ** exception exit: killed

Oops, look at that. It seems like this one is actually impossible to trap. Let's check something.

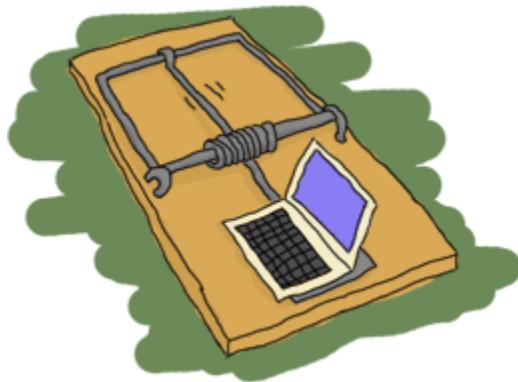
Exception source: `spawn_link(fun() -> exit(kill) end)`

Untrapped Result: ** exception exit: killed

Trapped Result: {'EXIT', <0.67.0>, kill}

Now that's getting confusing. When another process kills itself with `exit(kill)` and we don't trap exits, our own process dies with the reason killed. However, when we trap exits, things don't happen that way.

While you can trap most exit reasons, there are situations where you might want to brutally murder a process: maybe one of them is trapping exits but is also stuck in an infinite loop, never reading any message. The kill reason acts as a special signal that can't be trapped. This ensures any process you terminate with it will really be dead. Usually, kill is a bit of a last resort, when everything else has failed.



As the kill reason can never be trapped, it needs to be changed to killed when other processes receive the message. If it weren't changed in that manner, every other process linked to it would in turn die for the same kill reason and would in turn kill its neighbors, and so on. A death cascade would ensue.

This also explains why `exit(kill)` looks like `killed` when received from another linked process (the signal is modified so it doesn't cascade), but still looks like `kill` when trapped locally.

If you find this all confusing, don't worry. Many programmers feel the same. Exit signals are a bit of a funny beast. Luckily there aren't many more special cases than the ones described above. Once you understand those, you can understand most of Erlang's concurrent error management without a problem.

Monitors

So yeah. Maybe murdering processes isn't what you want. Maybe you don't feel like taking the world down with you

once you're gone. Maybe you're more of a stalker. In that case, monitors might be what you want.

More seriously, monitors are a special type of link with two differences:

- they are unidirectional;
- they can be stacked.



Monitors are what you want when a process wants to know what's going on with a second process, but neither of them really are vital to each other.

Another reason, as listed above, is stacking the references. Now this might seem useless from a quick look, but it is great

for writing libraries which need to know what's going on with other processes.

You see, links are more of an organizational construct. When you design the architecture of your application, you determine which process will do which jobs, and what will depend on what. Some processes will supervise others, some couldn't live without a twin process, etc. This structure is usually something fixed, known in advance. Links are useful for that and should not necessarily be used outside of it.

But what happens if you have 2 or 3 different libraries that you call and they all need to know whether a process is alive or not? If you were to use links for this, you would quickly hit a problem whenever you needed to unlink a process. Now, links aren't stackable, so the moment you unlink one, you unlink them all and mess up all the assumptions put up by the other libraries. That's pretty bad. So you need stackable links, and monitors are your solution. They can be removed individually. Plus, being unidirectional is handy in libraries because other processes shouldn't have to be aware of said libraries.

So what does a monitor look like? Easy enough, let's set one up. The function is erlang:monitor/2, where the first argument is the atom process and the second one is the pid:

```
1> erlang:monitor(process, spawn(fun() -> timer:sleep(500) end)).  
#Ref<0.0.0.77>  
2> flush().
```

```
shell got {'DOWN',#Ref<0.0.0.77>,process,<0.63.0>,normal}
ok
```

Every time a process you monitor goes down, you will receive such a message. The message is {'DOWN', MonitorReference, process, Pid, Reason}. The reference is there to allow you to demonitor the process. Remember, monitors are stackable, so it's possible to take more than one down. References allow you to track each of them in a unique manner. Also note that as with links, there is an atomic function to spawn a process while monitoring it, spawn_monitor/1-3:

```
3> {Pid, Ref} = spawn_monitor(fun() -> receive _ -> exit(boom) end end).
{<0.73.0>,#Ref<0.0.0.100>}
4> erlang:demonitor(Ref).
true
5> Pid ! die.
die
6> flush().
ok
```

In this case, we demonitored the other process before it crashed and as such we had no trace of it dying. The function demonitor/2 also exists and gives a little bit more information. The second parameter can be a list of options. Only two exist, info and flush:

```
7> f().
ok
8> {Pid, Ref} = spawn_monitor(fun() -> receive _ -> exit(boom) end end).
{<0.35.0>,#Ref<0.0.0.35>}
9> Pid ! die.
```

```
die
10> erlang:demonitor(Ref, [flush, info]).  
false
11> flush().  
ok
```

The `info` option tells you if a monitor existed or not when you tried to remove it. This is why the expression 10 returned `false`. Using `flush` as an option will remove the `DOWN` message from the mailbox if it existed, resulting in `flush()` finding nothing in the current process' mailbox.

Naming Processes

With links and monitors understood, there is another problem still left to be solved. Let's use the following functions of the `linkmon.erl` module:

```
start_critic() ->
    spawn(?MODULE, critic, []).  
  
judge(Pid, Band, Album) ->
    Pid ! {self(), {Band, Album}},
    receive
        {Pid, Criticism} -> Criticism
    after 2000 ->
        timeout
    end.  
  
critic() ->
    receive
        {From, {"Rage Against the Turing Machine", "Unit Testify"}} ->
            From ! {self(), "They are great!"};
```

```

{From, {"System of a Downtime", "Memoize"} } ->
  From ! {self(), "They're not Johnny Crash but they're good."};
{From, {"Johnny Crash", "The Token Ring of Fire"} } ->
  From ! {self(), "Simply incredible."};
{From, {_Band, _Album}} ->
  From ! {self(), "They are terrible!"}
end,
critic().

```

Now we'll just pretend we're going around stores, shopping for music. There are a few albums that sound interesting, but we're never quite sure. You decide to call your friend, the critic.

```

1> c(linkmon).
{ok,linkmon}
2> Critic = linkmon:start_critic().
<0.47.0>
3> linkmon:judge(Critic, "Genesis", "The Lambda Lies Down on Broadway").
"They are terrible!"

```

Because of a solar storm (I'm trying to find something realistic here), the connection is dropped:

```

4> exit(Critic, solar_storm).
true
5> linkmon:judge(Critic, "Genesis", "A trick of the Tail Recursion").
timeout

```

Annoying. We can no longer get criticism for the albums. To keep the critic alive, we'll write a basic 'supervisor' process whose only role is to restart it when it goes down:

```

start_critic2() ->
    spawn(?MODULE, restarter, []).

restarter() ->
    process_flag(trap_exit, true),
    Pid = spawn_link(?MODULE, critic, []),
    receive
        {'EXIT', Pid, normal} -> % not a crash
            ok;
        {'EXIT', Pid, shutdown} -> % manual termination, not a crash
            ok;
        {'EXIT', Pid, _} ->
            restarter()
    end.

```

Here, the restarter will be its own process. It will in turn start the critic's process and if it ever dies of abnormal cause, restarter/0 will loop and create a new critic. Note that I added a clause for {'EXIT', Pid, shutdown} as a way to manually kill the critic if we ever need to.

The problem with our approach is that there is no way to find the Pid of the critic, and thus we can't call him to have his opinion. One of the solutions Erlang has to solve this is to give names to processes.

The act of giving a name to a process allows you to replace the unpredictable pid by an atom. This atom can then be used exactly as a Pid when sending messages. To give a process a name, the function erlang:register/2 is used. If the process dies, it will automatically lose its name or you can also use unregister/1 to do it manually. You can get a list of

all registered processes with registered/0 or a more detailed one with the shell command regs(). Here we can rewrite the restarter/0 function as follows:

```
restarter() ->
    process_flag(trap_exit, true),
    Pid = spawn_link(?MODULE, critic, []),
    register(critic, Pid),
    receive
        {'EXIT', Pid, normal} -> % not a crash
            ok;
        {'EXIT', Pid, shutdown} -> % manual termination, not a crash
            ok;
        {'EXIT', Pid, _} ->
            restarter()
    end.
```

So as you can see, register/2 will always give our critic the name 'critic', no matter what the Pid is. What we need to do is then remove the need to pass in a Pid from the abstraction functions. Let's try this one:

```
judge2(Band, Album) ->
    critic ! {self(), {Band, Album}},
    Pid = whereis(critic),
    receive
        {Pid, Criticism} -> Criticism
    after 2000 ->
        timeout
    end.
```

Here, the line Pid = whereis(critic) is used to find the critic's process identifier in order to pattern match against it in the

receive expression. We want to match with this pid, because it makes sure we will match on the right message (there could be 500 of them in the mailbox as we speak!) This can be the source of a problem though. The code above assumes that the critic's pid will remain the same between the first two lines of the function. However, it is completely plausible the following will happen:

1. critic ! Message
2. critic receives
3. critic replies
4. critic dies
5. whereis fails
6. critic is restarted
7. code crashes

Or yet, this is also a possibility:

1. critic ! Message
2. critic receives
3. critic replies
4. critic dies
5. critic is restarted
6. whereis picks up
wrong pid
7. message never matches

The possibility that things go wrong in a different process can make another one go wrong if we don't do things right. In this case, the value of the `critic` atom can be seen from multiple processes. This is known as *shared state*. The problem here is that the value of `critic` can be accessed and

modified by different processes at virtually the same time, resulting in inconsistent information and software errors. The common term for such things is a *race condition*. Race conditions are particularly dangerous because they depend on the timing of events. In pretty much every concurrent and parallel language out there, this timing depends on unpredictable factors such as how busy the processor is, where the processes go, and what data is being processed by your program.

Don't drink too much kool-aid:

You might have heard that Erlang is usually free of race conditions or deadlocks and makes parallel code safe. This is true in many circumstances, but never assume your code is really that safe. Named processes are only one example of the multiple ways in which parallel code can go wrong.

Other examples include access to files on the computer (to modify them), updating the same database records from many different processes, etc.

Luckily for us, it's relatively easy to fix the code above if we don't assume the named process remains the same. Instead, we'll use references (created with `make_ref()`) as unique values to identify messages. We'll need to rewrite the `critic/0` function into `critic2/0` and `judge/3` into `judge2/2`:

```
judge2(Band, Album) ->
    Ref = make_ref(),
    critic ! {self(), Ref, {Band, Album}},
```

```
receive
    {Ref, Criticism} -> Criticism
after 2000 ->
    timeout
end.
```

```
critic2() ->
receive
    {From, Ref, {"Rage Against the Turing Machine", "Unit Testify"}} ->
        From ! {Ref, "They are great!"};
    {From, Ref, {"System of a Downtime", "Memoize"}} ->
        From ! {Ref, "They're not Johnny Crash but they're good."};
    {From, Ref, {"Johnny Crash", "The Token Ring of Fire"}} ->
        From ! {Ref, "Simply incredible."};
    {From, Ref, {_Band, _Album}} ->
        From ! {Ref, "They are terrible!"}
end,
critic2().
```

And then change `restarter/0` to fit by making it spawn `critic2/0` rather than `critic/0`. Now the other functions should keep working fine. The user won't see a difference. Well, they will because we renamed functions and changed the number of parameters, but they won't know what implementation details were changed and why it was important. All they'll see is that their code got simpler and they no longer need to send a pid around function calls:

```
6> c(linkmon).
{ok,linkmon}
7> linkmon:start_critic2().
<0.55.0>
8> linkmon:judge2("The Doors", "Light my Firewall").
```

```
"They are terrible!"  
9> exit(whereis(critic), kill).  
true  
10> linkmon:judge2("Rage Against the Turing Machine", "Unit Testify").  
"They are great!"
```

And now, even though we killed the critic, a new one instantly came back to solve our problems. That's the usefulness of named processes. Had you tried to call `linkmon:judge/2` without a registered process, a bad argument error would have been thrown by the `!` operator inside the function, making sure that processes that depend on named ones can't run without them.

Note: If you remember earlier texts, atoms can be used in a limited (though high) number. You shouldn't ever create dynamic atoms. This means naming processes should be reserved to important services unique to an instance of the VM and processes that should be there for the whole time your application runs.

If you need named processes but they are transient or there isn't any of them which can be unique to the VM, it may mean they need to be represented as a group instead. Linking and restarting them together if they crash might be the sane option, rather than trying to use dynamic names.

In the next chapter, we'll put the recent knowledge we gained on concurrent programming with Erlang to practice by writing a real application.

Designing a Concurrent Application



All fine and dandy. You understand the concepts, but then again, all we've had since the beginning of the book were toy examples: calculators, trees, Heathrow to London, etc. It's time for something more fun and more educational. We'll write a small application in concurrent Erlang. The application's going to be small and line-based, but still useful and moderately extensible.

I'm a somewhat unorganized person. I'm lost with homework, things to do around the apartment, this book, work, meetings, appointments, etc. I end up having a dozen of lists everywhere with tasks I still forget to do or look over. Hopefully you still need reminders of what to do (but you don't have a mind that wanders as much as mine does), because we're going to write one of these event reminder applications that prompt you to do stuff and remind you about appointments.

Understanding the Problem

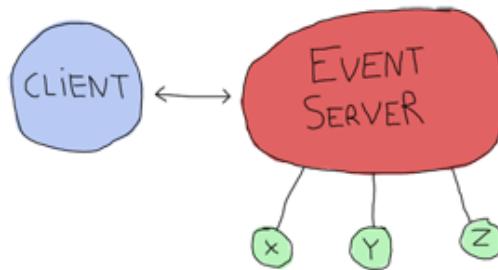
The first step is to know what the hell we're doing. "A reminder app," you say. "Of course," I say. But there's more. How do we plan

on interacting with the software? What do we want it to do for us? How do we represent the program with processes? How do we know what messages to send?

As the quote goes, "*Walking on water and developing software from a specification are easy if both are frozen.*" So let's get a spec and stick to it. Our little piece of software will allow us to do the following:

- Add an event. Events contain a deadline (the time to warn at), an event name and a description.
- Show a warning when the time has come for it.
- Cancel an event by name.
- No persistent disk storage. It's not needed to show the architectural concepts we'll see. It will suck for a real app, but I'll instead just show where it could be inserted if you wanted to do it and also point to a few helpful functions.
- Given we have no persistent storage, we have to be able to update the code while it is running.
- The interaction with the software will be done via the command line, but it should be possible to later extend it so other means could be used (say a GUI, web page access, instant messaging software, email, etc.)

Here's the structure of the program I picked to do it:



Where the client, event server and x, y and z are all processes.

Here's what each of them can do:

Event Server

- Accepts subscriptions from clients
- Forwards notifications from event processes to each of the subscribers
- Accepts messages to add events (and start the x, y, z processes needed)
- Can accept messages to cancel an event and subsequently kill the event processes
- Can be terminated by a client
- Can have its code reloaded via the shell.

client

- Subscribes to the event server and receive notifications as messages. As such it should be easy to design a bunch of clients all subscribing to the event server. Each of these could potentially be a gateway to the different interaction points mentioned above (GUI, web page, instant messaging software, email, etc.)
- Asks the server to add an event with all its details
- Asks the server to cancel an event
- Monitors the server (to know if it goes down)
- Shuts down the event server if needed

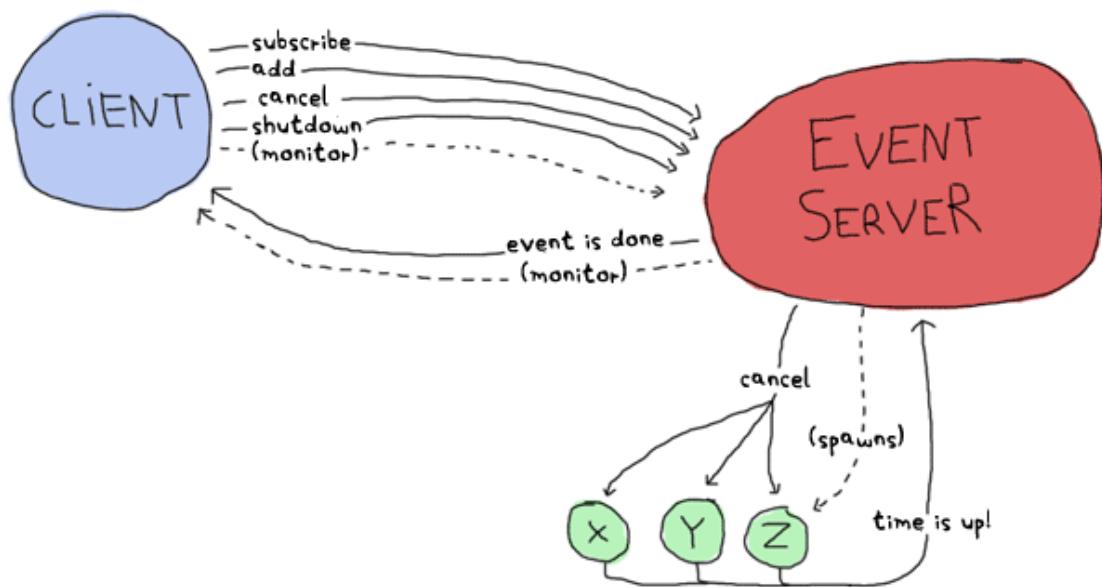
x, y and z:

- Represent a notification waiting to fire (they're basically just timers linked to the event server)
- Send a message to the event server when the time is up

- Receive a cancellation message and die

Note that all clients (IM, mail, etc. which are not implemented in this book) are notified about all events, and a cancellation is not something to warn the clients about. Here the software is written for you and me, and it's assumed only one user will run it.

Here's a more complex graph with all the possible messages:

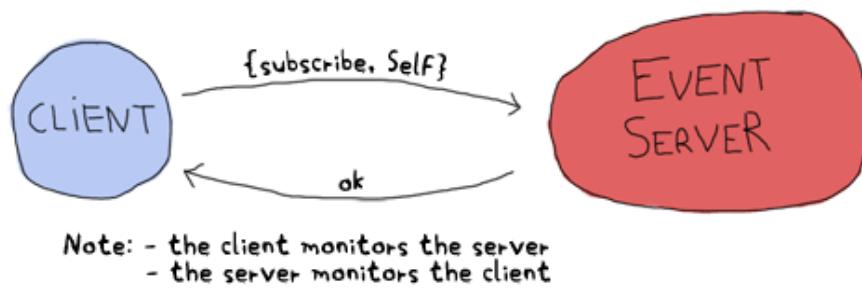


This represents every process we'll have. By drawing all the arrows there and saying they're messages, we've written a high level protocol, or at least its skeleton.

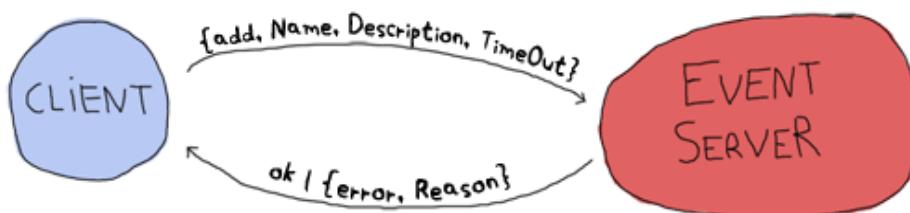
It should be noted that using one process per event to be reminded of is likely going to be overkill and hard to scale in a real world application. However, for an application you are going to be the sole user of, this is good enough. A different approach could be using functions such as `timer:send_after/2-3` to avoid spawning too many processes.

Defining the Protocol

Now that we know what each component has to do and to communicate, a good idea would be to make a list of all messages that will be sent and specify what they will look like. Let's first start with the communication between the client and the event server:

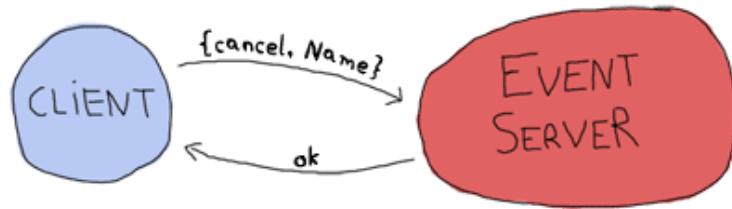


Here I chose to use two monitors because there is no obvious dependency between the client and the server. I mean, of course the client doesn't work without the server, but the server can live without a client. A link could have done the job right here, but because we want our system to be extensible with many clients, we can't assume other clients will all want to crash when the server dies. And neither can we assume the client can really be turned into a system process and trap exits in case the server dies. Now to the next message set:

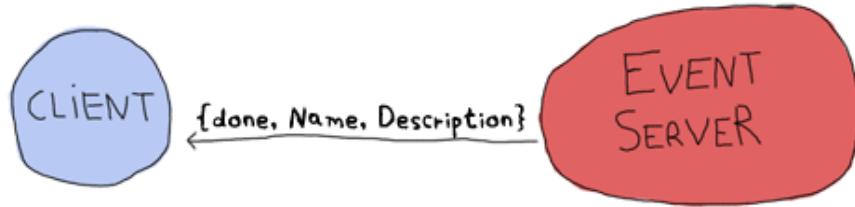


This adds an event to the event server. A confirmation is sent back under the form of the `ok` atom, unless something goes

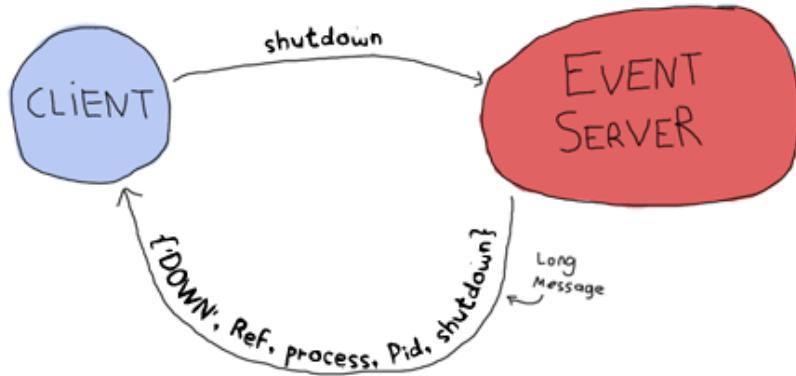
wrong (maybe the TimeOut is in the wrong format.) The inverse operation, removing events, can be done as follows:



The event server can then later send a notification when the event is due:



Then we only need the two following special cases for when we want to shut the server down or when it crashes:

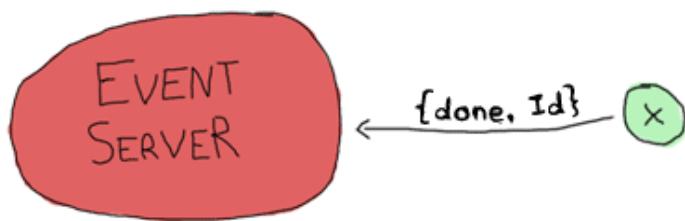


No direct confirmation is sent when the server dies because the monitor will already warn us of that. That's pretty much all that will happen between the client and the event server. Now for the

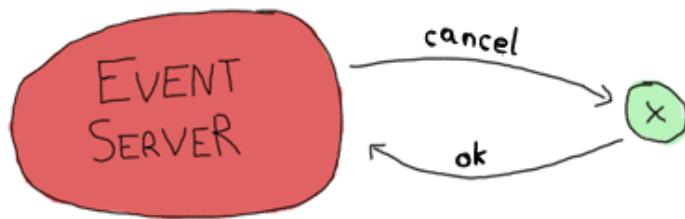
messages between the event server and the event processes themselves.

A thing to note here before we start is that it would be very useful to have the event server linked to the events. The reason for this is that we want all events to die if the server does: they make no sense without it.

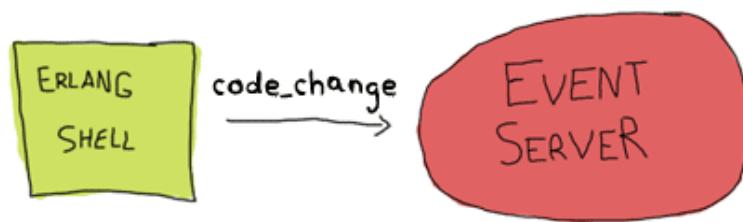
Ok, so back to the events. When the event server starts them, it gives each of them a special identifier (the event's name). Once one of these events' time has come, it needs to send a message saying so:



On the other hand, the event has to watch for cancel calls from the event server:



And that should be it. One last message will be needed for our protocol, the one that lets us upgrade the server:



No reply is necessary. We'll see why when we actually program that feature and you'll see it makes sense.

Having both the protocol defined and the general idea of how our process hierarchy will look in place, we can actually start working on the project.

Lay Them Foundations



To begin with it all, we should lay down a standard Erlang directory structure, which looks like this:

```
ebin/  
include/  
priv/  
src/
```

The `ebin/` directory is where files will go once they are compiled. The `include/` directory is used to store `.hrl` files that are to be included by other applications; the private `.hrl` files are usually just

kept inside the `src/` directory. The `priv/` directory is used for executables that might have to interact with Erlang, such as specific drivers and whatnot. We won't actually use that directory for this project. Then the last one is the `src/` directory, where all `.erl` files stay.

In standard Erlang projects, this directory structure can vary a little. A `conf/` directory can be added for specific configuration files, `doc/` for documentation and `lib/` for third party libraries required for your application to run. Different Erlang product on the market often use different names than these, but the four ones mentioned above usually stay the same given they're part of the standard OTP practices.

An Event Module

Get into the `src/` directory and start an `event.erl` module, which will implement the `x`, `y` and `z` events in the earlier drawings. I'm starting with this module because it's the one with the fewest dependencies: we'll be able to try to run it without needing to implement the event server or client functions.

Before really writing code, I have to mention that the protocol is incomplete. It helps represent what data will be sent from process to process, but not the intricacies of it: how the addressing works, whether we use references or names, etc. Most messages will be wrapped under the form `{Pid, Ref, Message}`, where `Pid` is the sender and `Ref` is a unique message identifier to help know what reply came from who. If we were to send many messages before looking for replies, we would not know what reply went with what message without a reference.

So here we go. The core of the processes that will run `event.erl`'s code will be the function `loop/1`, which will look a bit like the following skeleton if you remember the protocol:

```
loop(State) ->
    receive
        {Server, Ref, cancel} ->
            ...
        after Delay ->
            ...
    end.
```

This shows the timeout we have to support to announce an event has come to term and the way a server can call for the cancellation of an event. You'll notice a variable `State` in the loop. The `State` variable will have to contain data such as the timeout value (in seconds) and the name of the event (in order to send the message `{done, Id}`.) It will also need to know the event server's pid in order to send it notifications.

This is all stuff that's fit to be held in the loop's state. So let's declare a `state` record on the top of the file:

```
-module(event).
-compile(export_all).
-record(state, {server,
                name = "",
                to_go = 0}).
```

With this state defined, it should be possible to refine the loop a bit more:

```
loop(S = #state{server=Server}) ->
    receive
        {Server, Ref, cancel} ->
```

```
    Server ! {Ref, ok}
after S#state.to_go*1000 ->
    Server ! {done, S#state.name}
end.
```

Here, the multiplication by a thousand is to change the `to_go` value from seconds to milliseconds.

Don't drink too much Kool-Aid:

Language wart ahead! The reason why I bind the variable 'Server' in the function head is because it's used in pattern matching in the receive section. Remember, [records are hacks!](#) The expression `s#state.server` is secretly expanded to `element(2, s)`, which isn't a valid pattern to match on.

This still works fine for `s#state.to_go` after the `after` part, because that one can be an expression left to be evaluated later.

Now to test the loop:

```
6> c(event).
{ok,event}
7> rr(event, state).
[state]
8> spawn(event, loop, [#state{server=self(), name="test", to_go=5}]).
<0.60.0>
9> flush().
ok
10> flush().
Shell got {done,"test"}
ok
11> Pid = spawn(event, loop, [#state{server=self(), name="test", to_go=500}]).
<0.64.0>
12> ReplyRef = make_ref().
#Ref<0.0.0.210>
13> Pid ! {self(), ReplyRef, cancel}.
```

```
{<0.50.0>,#Ref<0.0.0.210>,cancel}  
14> flush().  
Shell got {#Ref<0.0.0.210>,ok}  
ok
```

Lots of stuff to see here. Well first of all, we import the record from the event module with `rr(Mod)`. Then, we spawn the event loop with the shell as the server (`self()`). This event should fire after 5 seconds. The 9th expression was run after 3 seconds, and the 10th one after 6 seconds. You can see we did receive the `{done, "test"}` message on the second try.

Right after that, I try the cancel feature (with an ample 500 seconds to type it). You can see I created the reference, sent the message and got a reply with the same reference so I know the `ok` I received was coming from this process and not any other on the system.

The reason why the cancel message is wrapped with a reference but the `done` message isn't is simply because we don't expect it to come from anywhere specific (any place will do, we won't match on the receive) nor should we want to reply to it. There's another test I want to do beforehand. What about an event happening next year?

```
15> spawn(event,loop,[#state{server=self(),name="test",to_go=365*24*60*60}]).  
<0.69.0>  
16>  
=ERROR REPORT===== DD-MM-YYYY::HH:mm:ss ===  
Error in process <0.69.0> with exit value: {timeout_value,[{event,loop,1}]}
```

Ouch. It seems like we hit an implementation limit. It turns out Erlang's timeout value is limited to about 50 days in milliseconds.

It might not be significant, but I'm showing this error for three reasons:

1. It bit me in the ass when writing the module and testing it, halfway through the chapter.
2. Erlang is certainly not perfect for every task and what we're seeing here is the consequences of using timers in ways not intended by the implementers.
3. That's not really a problem; let's work around it.

The fix I decided to apply for this one was to write a function that would split the timeout value into many parts if turns out to be too long. This will request some support from the `loop/1` function too. So yeah, the way to split the time is basically divide it in equal parts of 49 days (because the limit is about 50), and then put the remainder with all these equal parts. The sum of the list of seconds should now be the original time:

```
%% Because Erlang is limited to about 49 days (49*24*60*60*1000) in
%% milliseconds, the following function is used
normalize(N) ->
    Limit = 49*24*60*60,
    [N rem Limit | lists:duplicate(N div Limit, Limit)].
```

The function `lists:duplicate/2` will take a given expression as a second argument and reproduce it as many times as the value of the first argument (`[a,a,a] = lists:duplicate(3, a)`). If we were to send `normalize/1` the value `98*24*60*60+4`, it would return `[4,4233600,4233600]`. The `loop/1` function should now look like this to accommodate the new format:

```
%% Loop uses a list for times in order to go around the ~49 days limit
%% on timeouts.
loop(S = #state{server=Server, to_go=[T|Next]}) ->
```

```

receive
  {Server, Ref, cancel} ->
    Server ! {Ref, ok}
after T*1000 ->
  if Next == [] ->
    Server ! {done, S#state.name};
  Next /= [] ->
    loop(S#state{to_go=Next})
  end
end.

```

You can try it, it should work as normal, but now support years and years of timeout. How this works is that it takes the first element of the `to_go` list and waits for its whole duration. When this is done, the next element of the timeout list is verified. If it's empty, the timeout is over and the server is notified of it. Otherwise, the loop keeps going with the rest of the list until it's done.

It would be very annoying to have to manually call something like `event:normalize(N)` every time an event process is started, especially since our workaround shouldn't be of concern to programmers using our code. The standard way to do this is to instead have an `init` function handling all initialization of data required for the loop function to work well. While we're at it, we'll add the standard `start` and `start_link` functions:

```

start(EventName, Delay) ->
  spawn(?MODULE, init, [self(), EventName, Delay]).

start_link(EventName, Delay) ->
  spawn_link(?MODULE, init, [self(), EventName, Delay]).
```

```

%%% Event's innards
init(Server, EventName, Delay) ->
  loop(#state{server=Server,
```

```
name=EventName,  
to_go=normalize(Delay)}).
```

The interface is now much cleaner. Before testing, though, it would be nice to have the only message we can send, cancel, also have its own interface function:

```
cancel(Pid) ->  
    %% Monitor in case the process is already dead  
    Ref = erlang:monitor(process, Pid),  
    Pid ! {self(), Ref, cancel},  
    receive  
        {Ref, ok} ->  
            erlang:demonitor(Ref, [flush]),  
            ok;  
        {'DOWN', Ref, process, Pid, _Reason} ->  
            ok  
    end.
```

Oh! A new trick! Here I'm using a monitor to see if the process is there or not. If the process is already dead, I avoid useless waiting time and return `ok` as specified in the protocol. If the process replies with the reference, then I know it will soon die: I remove the reference to avoid receiving them when I no longer care about them. Note that I also supply the `flush` option, which will purge the DOWN message if it was sent before we had the time to demonitor.

Let's test these:

```
17> c(event).  
{ok,event}  
18> f().  
ok  
19> event:start("Event", 0).  
<0.103.0>  
20> flush().
```

```
Shell got {done,"Event"}
ok
21> Pid = event:start("Event", 500).
<0.106.0>
22> event:cancel(Pid).
ok
```

And it works! The last thing annoying with the event module is that we have to input the time left in seconds. It would be much better if we could use a standard format such as Erlang's `datetime` (`{Year, Month, Day}, {Hour, Minute, Second}`). Just add the following function that will calculate the difference between the current time on your computer and the delay you inserted:

```
time_to_go(TimeOut={{_,_,_},{_,_,_}}) ->
    Now = calendar:local_time(),
    ToGo = calendar:datetime_to_gregorian_seconds(TimeOut) -
            calendar:datetime_to_gregorian_seconds(Now),
    Secs = if ToGo > 0 -> ToGo;
            ToGo =< 0 -> 0
        end,
    normalize(Secs).
```

Oh, yeah. The calendar module has pretty funky function names. As noted above, this calculates the number of seconds between now and when the event is supposed to fire. If the event is in the past, we instead return 0 so it will notify the server as soon as it can. Now fix the init function to call this one instead of `normalize/1`. You can also rename `Delay` variables to say `DateTime` if you want the names to be more descriptive:

```
init(Server, EventName, DateTime) ->
    loop(#state{server=Server,
               name=EventName,
               to_go=time_to_go(DateTime)}).
```

Now that this is done, we can take a break. Start a new event, go drink a pint (half-litre) of milk/beer and come back just in time to see the event message coming in.

The Event Server

Let's deal with the event server. According to the protocol, the skeleton for that one should look a bit like this:

```
-module(evserv).  
-compile(export_all).  
  
loop(State) ->  
    receive  
        {Pid, MsgRef, {subscribe, Client}} ->  
            ...  
        {Pid, MsgRef, {add, Name, Description, TimeOut}} ->  
            ...  
        {Pid, MsgRef, {cancel, Name}} ->  
            ...  
        {done, Name} ->  
            ...  
        shutdown ->  
            ...  
        {'DOWN', Ref, process, _Pid, _Reason} ->  
            ...  
        code_change ->  
            ...  
        Unknown ->  
            io:format("Unknown message: ~p~n",[Unknown]),  
            loop(State)  
    end.
```

You'll notice I have wrapped calls that require replies with the same {Pid, Ref, Message} format as earlier. Now, the server will need to keep two things in its state: a list of subscribing clients and a

list of all the event processes it spawned. If you have noticed, the protocol says that when an event is done, the event server should receive {done, Name}, but send {done, Name, Description}. The idea here is to have as little traffic as necessary and only have the event processes care about what is strictly necessary. So yeah, list of clients and list of events:

```
-record(state, {events, %% list of #event{} records
               clients}). %% list of Pids

-record(event, {name = "",
                description = "",
                pid,
                timeout = {{1970,1,1},{0,0,0}}}).
```

And the loop now has the record definition in its head:

```
loop(S = #state{}) ->
    receive
        ...
    end.
```

It would be nice if both events and clients were orddicts. We're unlikely to have many hundreds of them at once. If you recall the chapter on [data structures](#), orddicts fit that need very well. We'll write an init function to handle this:

```
init() ->
    %% Loading events from a static file could be done here.
    %% You would need to pass an argument to init telling where the
    %% resource to find the events is. Then load it from here.
    %% Another option is to just pass the events straight to the server
    %% through this function.
    loop(#state{events=orddict:new(),
               clients=orddict:new()}).
```

With the skeleton and initialization done, I'll implement each message one by one. The first message is the one about subscriptions. We want to keep a list of all subscribers because when an event is done, we have to notify them. Also, the protocol above mentions we should monitor them. It makes sense because we don't want to hold onto crashed clients and send useless messages for no reason. Anyway, it should look like this:

```
{Pid, MsgRef, {subscribe, Client}} ->  
    Ref = erlang:monitor(process, Client),  
    NewClients = orddict:store(Ref, Client, S#state.clients),  
    Pid ! {MsgRef, ok},  
    loop(S#state{clients=NewClients});
```



So what this section of `loop/1` does is start a monitor, and store the client info in the orddict under the key `Ref`. The reason for this is simple: the only other time we'll need to fetch the client ID will be if we receive a monitor's `EXIT` message, which will contain the reference (which will let us get rid of the orddict's entry).

The next message to care about is the one where we add events. Now, it is possible to return an error status. The only validation we'll do is check the timestamps we accept. While it's easy to subscribe to the `{{Year,Month,Day}, {Hour,Minute,seconds}}` layout, we have to make sure we don't do things like accept events on February 29 when we're not in a leap year, or any other date that

doesn't exist. Moreover, we don't want to accept impossible date values such as "5 hours, minus 1 minute and 75 seconds". A single function can take care of validating all of that.

The first building block we'll use is the function `calendar:valid_date/1`. This one, as the name says, checks if the date is valid or not. Sadly, the weirdness of the calendar module doesn't stop at funky names: there is actually no function to confirm that `{H,M,S}` has valid values. We'll have to implement that one too, following the funky naming scheme:

```
valid_datetime({Date,Time}) ->
    try
        calendar:valid_date(Date) andalso valid_time(Time)
    catch
        error:function_clause -> %% not in {{Y,M,D},{H,Min,S}} format
            false
    end;
valid_datetime(_) ->
    false.
```

```
valid_time({H,M,S}) -> valid_time(H,M,S).
valid_time(H,M,S) when H >= 0, H < 24,
    M >= 0, M < 60,
    S >= 0, S < 60 -> true;
valid_time(_,_,_) -> false.
```

The `valid_datetime/1` function can now be used in the part where we try to add the message:

```
{Pid, MsgRef, {add, Name, Description, TimeOut}} ->
case valid_datetime(TimeOut) of
    true ->
        EventPid = event:start_link(Name, TimeOut),
        NewEvents = orddict:store(Name,
```

```

#event{name=Name,
        description=Description,
        pid=EventPid,
        timeout=TimeOut},
S#state.events),
Pid ! {MsgRef, ok},
loop(S#state{events=NewEvents});
false ->
    Pid ! {MsgRef, {error, bad_timeout}},
    loop(S)
end;

```

If the time is valid, we spawn a new event process, then store its data in the event server's state before sending a confirmation to the caller. If the timeout is wrong, we notify the client rather than having the error pass silently or crashing the server. Additional checks could be added for name clashes or other restrictions (just remember to update the protocol documentation!)

The next message defined in our protocol is the one where we cancel an event. Canceling an event never fails on the client side, so the code is simpler there. Just check whether the event is in the process' state record. If it is, use the `event:cancel/1` function we defined to kill it and send `ok`. If it's not found, just tell the user everything went right anyway -- the event is not running and that's what the user wanted.

```

{Pid, MsgRef, {cancel, Name}} ->
Events = case orddict:find(Name, S#state.events) of
    {ok, E} ->
        event:cancel(E#event.pid),
        orddict:erase(Name, S#state.events);
    error ->
        S#state.events
end,

```

```

Pid ! {MsgRef, ok},
loop(S#state{events=Events});

```

Good, good. So now all voluntary interaction coming from the client to the event server is covered. Let's deal with the stuff that's going between the server and the events themselves. There are two messages to handle: canceling the events (which is done), and the events timing out. That message is simply {done, Name}:

```

{done, Name} ->
  case orddict:find(Name, S#state.events) of
    {ok, E} ->
      send_to_clients({done, E#event.name, E#event.description},
                      S#state.clients),
      NewEvents = orddict:erase(Name, S#state.events),
      loop(S#state{events=NewEvents});
    error ->
      %% This may happen if we cancel an event and
      %% it fires at the same time
      loop(S)
  end;

```

And the function `send_to_clients/2` does as its name says and is defined as follows:

```

send_to_clients(Msg, ClientDict) ->
  orddict:map(fun(_Ref, Pid) -> Pid ! Msg end, ClientDict).

```

That should be it for most of the loop code. What's left is the set different status messages: clients going down, shutdown, code upgrades, etc. Here they come:

```

shutdown ->
  exit(shutdown);
{'DOWN', Ref, process, _Pid, _Reason} ->
  loop(S#state{clients=orddict:erase(Ref, S#state.clients)});
```

```
code_change ->
    ?MODULE:loop(s);
Unknown ->
    io:format("Unknown message: ~p~n",[Unknown]),
    loop(s)
```

The first case (`shutdown`) is pretty explicit. You get the kill message, let the process die. If you wanted to save state to disk, that could be a possible place to do it. If you wanted safer save/exit semantics, this could be done on every `add`, `cancel` or `done` message. Loading events from disk could then be done in the `init` function, spawning them as they come.

The '`DOWN`' message's actions are also simple enough. It means a client died, so we remove it from the client list in the state.

Unknown messages will just be shown with `io:format/2` for debugging purposes, although a real production application would likely use a dedicated logging module

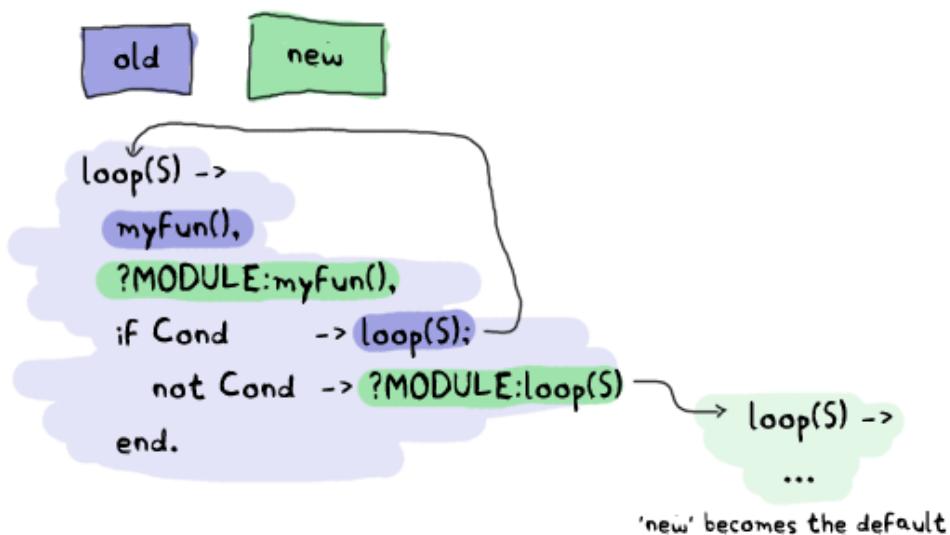
And here comes the code change message. This one is interesting enough for me to give it its own section.

Hot Code Loving

In order to do hot code loading, Erlang has a thing called the `code server`. The code server is basically a VM process in charge of an ETS table (in-memory database table, native to the VM.) The code server can hold two versions of a single module in memory, and both versions can run at once. A new version of a module is automatically loaded when compiling it with `c(Module)`, loading with `I(Module)` or loading it with one of the many functions of the `code` module.

A concept to understand is that Erlang has both *local* and *external* calls. Local calls are those function calls you can make with functions that might not be exported. They're just of the format Atom(Args). An external call, on the other hand, can only be done with exported functions and has the form Module:Function(Args).

When there are two versions of a module loaded in the VM, all local calls are done through the currently running version in a process. However, external calls are **always** done on the newest version of the code available in the code server. Then, if local calls are made from within the external one, they are in the new version of the code.



Given that every process/actor in Erlang needs to do a recursive call in order to change its state, it is possible to load entirely new versions of an actor by having an external recursive call.

Note: If you load a third version of a module while a process still runs with the first one, that process gets killed by the VM, which

assumes it was an orphan process without a supervisor or a way to upgrade itself. If nobody runs the oldest version, it is simply dropped and the newest ones are kept instead.

There are ways to bind yourself to a system module that will send messages whenever a new version of a module is loaded. By doing this, you can trigger a module reload only when receiving such a message, and always do it with a code upgrade function, say `MyModule:Upgrade(CurrentState)`, which will then be able to transform the state data structure according to the new version's specification. This 'subscription' handling is done automatically by the OTP framework, which we'll start studying soon enough. For the reminder application, we won't use the code server and will instead use a custom `code_change` message from the shell, doing very basic reloading. That's pretty much all you need to know to do hot code loading. Nevertheless, here's a more generic example:

```
-module(hotload).
-export([server/1, upgrade/1]).

server(State) ->
    receive
        update ->
            NewState = ?MODULE:upgrade(State),
            ?MODULE:server(NewState); %% loop in the new version of the module
        SomeMessage ->
            %% do something here
            server(State) %% stay in the same version no matter what.
    end.

upgrade(OldState) ->
    %% transform and return the state here.
```

As you can see, our ?MODULE:loop(s) fits this pattern.

I Said, Hide Your Messages

Hiding messages! If you expect people to build on your code and processes, you must hide the messages in interface functions.

Here's what we used for the evserv module:

```
start() ->
    register(?MODULE, Pid=spawn(?MODULE, init, [])),
    Pid.
```

```
start_link() ->
    register(?MODULE, Pid=spawn_link(?MODULE, init, [])),
    Pid.
```

```
terminate() ->
    ?MODULE ! shutdown.
```

I decided to register the server module because, for now, we should only have one running at a time. If you were to expand the reminder use to support many users, it would be a good idea to instead register the names with the global module, or the gproc library. For the sake of this example app, this will be enough.

The first message we wrote is the next we should abstract away: how to subscribe. The little protocol or specification I wrote above called for a monitor, so this one is added there. At any point, if the reference returned by the subscribe message is in a DOWN message, the client will know the server has gone down.

```
subscribe(Pid) ->
    Ref = erlang:monitor(process, whereis(?MODULE)),
    ?MODULE ! {self(), Ref, {subscribe, Pid}},
    receive
```

```

{Ref, ok} ->
    {ok, Ref};
{'DOWN', Ref, process, _Pid, Reason} ->
    {error, Reason}
after 5000 ->
    {error, timeout}
end.

```

The next one is the event adding:

```

add_event(Name, Description, TimeOut) ->
    Ref = make_ref(),
    ?MODULE ! {self(), Ref, {add, Name, Description, TimeOut}},
receive
    {Ref, Msg} -> Msg
after 5000 ->
    {error, timeout}
end.

```

Note that I choose to forward the {error, bad_timeout} message that could be received to the client. I could have also decided to crash the client by raising erlang:error(bad_timeout). Whether crashing the client or forwarding the error message is the thing to do is still debated in the community. Here's the alternative crashing function:

```

add_event2(Name, Description, TimeOut) ->
    Ref = make_ref(),
    ?MODULE ! {self(), Ref, {add, Name, Description, TimeOut}},
receive
    {Ref, {error, Reason}} -> erlang:error(Reason);
    {Ref, Msg} -> Msg
after 5000 ->
    {error, timeout}
end.

```

Then there's event cancellation, which just takes a name:

```
cancel(Name) ->
    Ref = make_ref(),
    ?MODULE ! {self(), Ref, {cancel, Name}},
    receive
        {Ref, ok} -> ok
    after 5000 ->
        {error, timeout}
    end.
```

Last of all is a small nicety provided for the client, a function used to accumulate all messages during a given period of time. If messages are found, they're all taken and the function returns as soon as possible:

```
listen(Delay) ->
    receive
        M = {done, _Name, _Description} ->
            [M | listen(0)]
    after Delay*1000 ->
        []
    end.
```

A Test Drive

You should now be able to compile the application and give it a test run. To make things a bit simpler, we'll write a specific Erlang makefile to build the project. Open a file named `Emakefile` and put it in the project's base directory. The file contains Erlang terms and gives the Erlang compiler the recipe to cook wonderful and crispy `.beam` files:



```
{'src/*', [debug_info,  
          {i, "src"},  
          {i, "include"},  
          {outdir, "ebin"}]}.
```

This tells the compiler to add `debug_info` to the files (this is rarely an option you want to give up), to look for files in the `src/` and `include/` directory and to output them in `ebin/`.

Now, by going in your command line and running `erl -make`, the files should all be compiled and put inside the `ebin/` directory for you. Start the Erlang shell by doing `erl -pa ebin/`. The `-pa <directory>` option tells the Erlang VM to add that path to the places it can look in for modules.

Another option is to start the shell as usual and call `make:all([load])`. This will look for a file named 'Emakefile' in the current directory, recompile it (if it changed) and load the new files.

You should now be able to track thousands of events (just replace the `DateTime` variables with whatever makes sense when you're writing the text):

```
1> evserv:start().  
<0.34.0>
```

```

2> evserv:subscribe(self()).
{ok,#Ref<0.0.0.31>}
3> evserv:add_event("Hey there", "test", FutureDateTime).
ok
4> evserv:listen(5).
[]
5> evserv:cancel("Hey there").
ok
6> evserv:add_event("Hey there2", "test", NextMinuteDateTime).
ok
7> evserv:listen(2000).
[{done,"Hey there2","test"}]

```

Nice nice nice. Writing any client should now be simple enough given the few basic interface functions we have created.

Adding Supervision

In order to be a more stable application, we should write another 'restarter' as we did in the [last chapter](#). Open up a file named sup.erl where our supervisor will be:

```

-module(sup).
-export([start/2, start_link/2, init/1, loop/1]).
```

```

start(Mod,Args) ->
    spawn(?MODULE, init, [{Mod, Args}]).
```

```

start_link(Mod,Args) ->
    spawn_link(?MODULE, init, [{Mod, Args}]).
```

```

init({Mod,Args}) ->
    process_flag(trap_exit, true),
    loop({Mod,start_link,Args}).
```

```

loop({M,F,A}) ->
```

```

Pid = apply(M,F,A),
receive
    {'EXIT', _From, shutdown} ->
        exit(shutdown); % will kill the child too
    {'EXIT', Pid, Reason} ->
        io:format("Process ~p exited for reason ~p~n",[Pid,Reason]),
        loop({M,F,A})
end.

```

This is somewhat similar to the 'restarter', although this one is a tad more generic. It can take any module, as long as it has a start_link function. It will restart the process it watches indefinitely, unless the supervisor itself is terminated with a shutdown exit signal. Here it is in use:

```

1> c(evserv), c(sup).
{ok,sup}
2> SupPid = sup:start(evserv, []).
<0.43.0>
3> whereis(evserv).
<0.44.0>
4> exit(whereis(evserv), die).
true
Process <0.44.0> exited for reason die
5> exit(whereis(evserv), die).
Process <0.48.0> exited for reason die
true
6> exit(SupPid, shutdown).
true
7> whereis(evserv).
undefined

```

As you can see, killing the supervisor will also kill its child.

Note: We'll see much more advanced and flexible supervisors in the chapter about OTP supervisors. Those are the ones people are

thinking of when they mention *supervision trees*. The supervisor demonstrated here is only the most basic form that exists and is not exactly fit for production environments compared to the real thing.

Namespaces (or lack thereof)



Because Erlang has a flat module structure (there is no hierarchy), it is frequent for some applications to enter in conflict. One example of this is the frequently used `user` module that almost every project attempts to define at least once. This clashes with the `user` module shipped with Erlang. You can test for any clashes with the function `code:clash/0`.

Because of this, the common pattern is to prefix every module name with the name of your project. In this case, our reminder application's modules should be renamed to `reminder_evserv`, `reminder_sup` and `reminder_event`.

Some programmers then decide to add a module, named after the application itself, which wraps common calls that

programmers could use when using their own application. Example calls could be functions such as starting the application with a supervisor, subscribing to the server, adding and cancelling events, etc.

It's important to be aware of other namespaces, too, such as registered names that must not clash, database tables, etc.

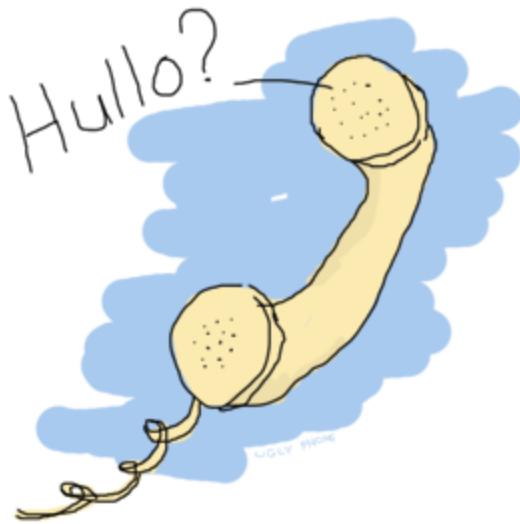
That's pretty much it for a very basic concurrent Erlang application. This one showed we could have a bunch of concurrent processes without thinking too hard about it: supervisors, clients, servers, processes used as timers (and we could have thousands of them), etc. No need to synchronize them, no locks, no real main loop. Message passing has made it simple to compartmentalize our application into a few modules with separated concerns and tasks.

The basic calls inside `evserv.erl` could now be used to construct clients that would allow to interact with the event server from somewhere outside of the Erlang VM and make the program truly useful.

Before doing that, though, I suggest you read up on the OTP framework. The next few chapters will cover some of its building blocks, which will allow for much more robust and elegant applications. A huge part of Erlang's power comes from using it. It's a carefully crafted and well-engineered tool that any self-respecting Erlang programmer has to know.

What is OTP?

It's The Open Telecom Platform!



OTP stands for *Open Telecom Platform*, although it's not that much about telecom anymore (it's more about software that has the property of telecom applications, but yeah.) If half of Erlang's greatness comes from its concurrency and distribution and the other half comes from its error handling capabilities, then the OTP framework is the third half of it.

During the previous chapters, we've seen a few examples of common practices on how to write concurrent applications with the languages' built-in facilities: links, monitors, servers, timeouts, trapping exits, etc. There were a few 'gotchas' here and there on the order things need to be done, on how to avoid race conditions or to always remember that a process

could die at any time. There was also hot code loading, naming processes and adding supervisors, to name a few.

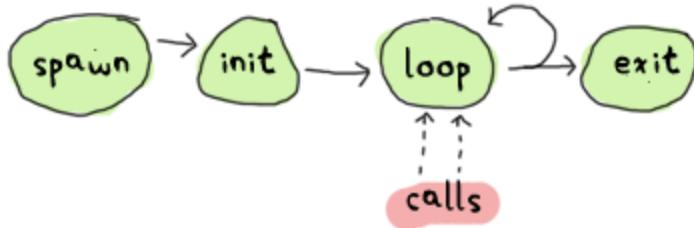
Doing all of this manually is time consuming and sometimes prone to error. There are corner cases to be forgotten about and pits to fall into. The OTP framework takes care of this by grouping these essential practices into a set of libraries that have been carefully engineered and battle-hardened over years. Every Erlang programmer should use them.

The OTP framework is also a set of modules and standards designed to help you build applications. Given most Erlang programmers end up using OTP, most Erlang applications you'll encounter in the wild will tend to follow these standards.

The Common Process, Abstracted

One of the things we've done many times in the previous process examples is divide everything in accordance to very specific tasks. In most processes, we had a function in charge of spawning the new process, a function in charge of giving it its initial values, a main loop, etc.

These parts, as it turns out, are usually present in all concurrent programs you'll write, no matter what the process might be used for.



The engineers and computer scientists behind the OTP framework spotted these patterns and included them in a bunch of common libraries. These libraries are built with code that is equivalent to most of the abstractions we used (like using references to tag messages), with the advantage of being used for years in the field and also being built with far more caution than we were with our implementations. They contain functions to safely spawn and initialize processes, send messages to them in a fault-tolerant manner and many other things. Funnily enough, you should rarely need to use these libraries yourself. The abstractions they contain are so basic and universal that a lot more interesting things were built on top of them. Those libraries are the ones we'll use.



In the following chapters we'll see a few of the common uses of processes and then how they can be abstracted, then made generic. Then for each of these we'll also see the corresponding implementation with the OTP framework's behaviours and how to use each of them.

The Basic Server

The first common pattern I'll describe is one we've already used. When writing the [event server](#), we had what could be called a *client-server model*. The event server would receive calls from the client, act on them and then reply to it if the protocol said to do so.

For this chapter, we'll use a very simple server, allowing us to focus on the essential properties of it. Here's the `kitty_server`:

```
%%%%% Naive version
-module(kitty_server).

-export([start_link/0, order_cat/4, return_cat/2, close_shop/1]).
```

```
-record(cat, {name, color=green, description}).
```

```
%%% Client API
start_link() -> spawn_link(fun init/0).
```

```
%% Synchronous call
order_cat(Pid, Name, Color, Description) ->
    Ref = erlang:monitor(process, Pid),
    Pid ! {self(), Ref, {order, Name, Color, Description}},
    receive
        {Ref, Cat} ->
            erlang:demonitor(Ref, [flush]),
            Cat;
        {'DOWN', Ref, process, Pid, Reason} ->
            erlang:error(Reason)
    after 5000 ->
        erlang:error(timeout)
    end.
```

```
%% This call is asynchronous
return_cat(Pid, Cat = #cat{}) ->
    Pid ! {return, Cat},
    ok.
```

```
%% Synchronous call
close_shop(Pid) ->
    Ref = erlang:monitor(process, Pid),
    Pid ! {self(), Ref, terminate},
    receive
```

```

{Ref, ok} ->
    erlang:demonitor(Ref, [flush]),
    ok;
{'DOWN', Ref, process, Pid, Reason} ->
    erlang:error(Reason)
after 5000 ->
    erlang:error(timeout)
end.

%%% Server functions
init() -> loop([]).

loop(Cats) ->
    receive
        {Pid, Ref, {order, Name, Color, Description}} ->
            if Cats =:= [] ->
                Pid ! {Ref, make_cat(Name, Color, Description)},
                loop(Cats);
            Cats =/= [] -> % got to empty the stock
                Pid ! {Ref, hd(Cats)},
                loop(tl(Cats))
            end;
        {return, Cat = #cat{}} ->
            loop([Cat|Cats]);
        {Pid, Ref, terminate} ->
            Pid ! {Ref, ok},
            terminate(Cats);
        Unknown ->
            %% do some logging here too
            io:format("Unknown message: ~p~n", [Unknown]),
            loop(Cats)
    end.

```

%%% Private functions

```

make_cat(Name, Col, Desc) ->
    #cat{name = Name, color = Col, description = Desc}.

terminate(Cats) ->
    [io:format("~p was set free.~n", [C#cat.name]) || C <- Cats],
    ok.

```

So this is a kitty server/store. The behavior is extremely simple: you describe a cat and you get that cat. If someone returns a cat, it's added to a list and is then automatically sent as the next order instead of what the client actually asked for (we're in this kitty store for the money, not smiles):

```

1> c(kitty_server).
{ok,kitty_server}
2> rr(kitty_server).
[cat]
3> Pid = kitty_server:start_link().
<0.57.0>
4> Cat1 = kitty_server:order_cat(Pid, carl, brown, "loves to burn bridges").
#cat{name = carl,color = brown,
     description = "loves to burn bridges"}
5> kitty_server:return_cat(Pid, Cat1).
ok
6> kitty_server:order_cat(Pid, jimmy, orange, "cuddly").
#cat{name = carl,color = brown,
     description = "loves to burn bridges"}
7> kitty_server:order_cat(Pid, jimmy, orange, "cuddly").
#cat{name = jimmy,color = orange,description = "cuddly"}
8> kitty_server:return_cat(Pid, Cat1).
ok
9> kitty_server:close_shop(Pid).
carl was set free.

```

```
ok
10> kitty_server:close_shop(Pid).
** exception error: no such process or port
   in function  kitty_server:close_shop/1
```

Looking back at the source code for the module, we can see patterns we've previously applied. The sections where we set monitors up and down, apply timers, receive data, use a main loop, handle the init function, etc. should all be familiar. It should be possible to abstract away these things we end up repeating all the time.

Let's first take a look at the client API. The first thing we can notice is that both synchronous calls are extremely similar. These are the calls that would likely go in abstraction libraries as mentioned in the previous section. For now, we'll just abstract these away as a single function in a new module which will hold all the generic parts of the kitty server:

```
-module(my_server).
-compile(export_all).

call(Pid, Msg) ->
    Ref = erlang:monitor(process, Pid),
    Pid ! {self(), Ref, Msg},
    receive
        {Ref, Reply} ->
            erlang:demonitor(Ref, [flush]),
            Reply;
        {'DOWN', Ref, process, Pid, Reason} ->
            erlang:error(Reason)
```

```
after 5000 ->
    erlang:error(timeout)
end.
```

This takes a message and a PID, sticks them into in the function, then forwards the message for you in a safe manner. From now on, we can just substitute the message sending we do with a call to this function. So if we were to rewrite a new kitty server to be paired with the abstracted my_server, it could begin like this:

```
-module(kitty_server2).
-export([start_link/0, order_cat/4, return_cat/2, close_shop/1]).

-record(cat, {name, color=green, description}).

%%% Client API
start_link() -> spawn_link(fun init/0).

%%% Synchronous call
order_cat(Pid, Name, Color, Description) ->
    my_server:call(Pid, {order, Name, Color, Description}).

%%% This call is asynchronous
return_cat(Pid, Cat = #cat{}) ->
    Pid ! {return, Cat},
    ok.

%%% Synchronous call
close_shop(Pid) ->
    my_server:call(Pid, terminate).
```

The next big generic chunk of code we have is not as obvious as the `call/2` function. Note that every process we've written so far has a loop where all the messages are pattern matched. This is a bit of a touchy part, but here we have to separate the pattern matching from the loop itself. One quick way to do it would be to add:

```
loop(Module, State) ->
    receive
        Message -> Module:handle(Message, State)
    end.
```

And then the specific module can look like this:

```
handle(Message1, State) -> NewState1;
handle(Message2, State) -> NewState2;
...
handle(MessageN, State) -> NewStateN.
```

This is better. There are still ways to make it even cleaner. If you paid attention when reading the `kitty_server` module (and I hope you did!), you will have noticed we have a specific way to call synchronously and another one to call asynchronously. It would be pretty helpful if our generic server implementation could provide a clear way to know which kind of call is which.

In order to do this, we will need to match different kinds of messages in `my_server:loop/2`. This means we'll need to change the `call/2` function a little bit so synchronous calls are made

obvious by adding the atom sync to the message on the function's second line:

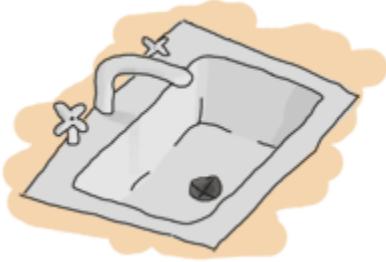
```
call(Pid, Msg) ->
    Ref = erlang:monitor(process, Pid),
    Pid ! {sync, self(), Ref, Msg},
    receive
        {Ref, Reply} ->
            erlang:demonitor(Ref, [flush]),
            Reply;
        {'DOWN', Ref, process, Pid, Reason} ->
            erlang:error(Reason)
    after 5000 ->
        erlang:error(timeout)
    end.
```

We can now provide a new function for asynchronous calls. The function cast/2 will handle this:

```
cast(Pid, Msg) ->
    Pid ! {async, Msg},
    ok.
```

With this done, the loop can now look like this:

```
loop(Module, State) ->
    receive
        {async, Msg} ->
            loop(Module, Module:handle_cast(Msg, State));
        {sync, Pid, Ref, Msg} ->
            loop(Module, Module:handle_call(Msg, Pid, Ref, State))
    end.
```



And then you could also add specific slots to handle messages that don't fit the sync/async concept (maybe they were sent by accident) or to have your debug functions and other stuff like hot code reloading in there.

One disappointing thing with the loop above is that the abstraction is leaking. The programmers who will use `my_server` will still need to know about references when sending synchronous messages and replying to them. That makes the abstraction useless. To use it, you still need to understand all the boring details. Here's a quick fix for it:

```
loop(Module, State) ->
    receive
        {async, Msg} ->
            loop(Module, Module:handle_cast(Msg, State));
        {sync, Pid, Ref, Msg} ->
            loop(Module, Module:handle_call(Msg, {Pid, Ref}, State))
    end.
```

By putting both variables `Pid` and `Ref` in a tuple, they can be passed as a single argument to the other function as a variable with a name like `From`. Then the user doesn't have to know anything about the variable's innards. Instead, we'll

provide a function to send replies that should understand what *From* contains:

```
reply({Pid, Ref}, Reply) ->
    Pid ! {Ref, Reply}.
```

What is left to do is specify the starter functions (`start`, `start_link` and `init`) that pass around the module names and whatnot. Once they're added, the module should look like this:

```
-module(my_server).
-export([start/2, start_link/2, call/2, cast/2, reply/2]).
```

```
%%% Public API
start(Module, InitialState) ->
    spawn(fun() -> init(Module, InitialState) end).
```

```
start_link(Module, InitialState) ->
    spawn_link(fun() -> init(Module, InitialState) end).
```

```
call(Pid, Msg) ->
    Ref = erlang:monitor(process, Pid),
    Pid ! {sync, self(), Ref, Msg},
    receive
        {Ref, Reply} ->
            erlang:demonitor(Ref, [flush]),
            Reply;
        {'DOWN', Ref, process, Pid, Reason} ->
            erlang:error(Reason)
    after 5000 ->
        erlang:error(timeout)
    end.
```

```
cast(Pid, Msg) ->
```

```

Pid ! {async, Msg},
ok.

reply({Pid, Ref}, Reply) ->
  Pid ! {Ref, Reply}.

%%% Private stuff
init(Module, InitialState) ->
  loop(Module, Module:init(InitialState)).

loop(Module, State) ->
  receive
    {async, Msg} ->
      loop(Module, Module:handle_cast(Msg, State));
    {sync, Pid, Ref, Msg} ->
      loop(Module, Module:handle_call(Msg, {Pid, Ref}, State))
  end.

The next thing to do is reimplement the kitty server, now
kitty_server2 as a callback module that will respect the
interface we defined for my_server. We'll keep the same
interface as the previous implementation, except all the calls
are now redirected to go through my_server:

-module(kitty_server2).

-export([start_link/0, order_cat/4, return_cat/2, close_shop/1]).
-export([init/1, handle_call/3, handle_cast/2]).

-record(cat, {name, color=green, description}).

%%% Client API
start_link() -> my_server:start_link(?MODULE, []).

```

```
%% Synchronous call
order_cat(Pid, Name, Color, Description) ->
    my_server:call(Pid, {order, Name, Color, Description}).
```

```
%% This call is asynchronous
return_cat(Pid, Cat = #cat{}) ->
    my_server:cast(Pid, {return, Cat}).
```

```
%% Synchronous call
close_shop(Pid) ->
    my_server:call(Pid, terminate).
```

Note that I added a second `-export()` at the top of the module. Those are the functions `my_server` will need to call to make everything work:

```
%%% Server functions
init([]) -> [] . %% no treatment of info here!

handle_call({order, Name, Color, Description}, From, Cats) ->
    if Cats == [] ->
        my_server:reply(From, make_cat(Name, Color, Description)),
        Cats;
    Cats /= [] ->
        my_server:reply(From, hd(Cats)),
        tl(Cats)
    end;

handle_call(terminate, From, Cats) ->
    my_server:reply(From, ok),
    terminate(Cats).
```

```
handle_cast({return, Cat = #cat{}}, Cats) ->
    [Cat|Cats].
```

And then what needs to be done is to re-add the private functions:

```
%%% Private functions
make_cat(Name, Col, Desc) ->
    #cat{name=Name, color=Col, description=Desc}.

terminate(Cats) ->
    [io:format("~p was set free.~n",[C#cat.name]) || C <- Cats],
    exit(normal).
```

Just make sure to replace the ok we had before by exit(normal) in terminate/1, otherwise the server will keep going on.

The code should be compilable and testable, and run in exactly the same manner as it was before. The code is quite similar, but let's see what changed.

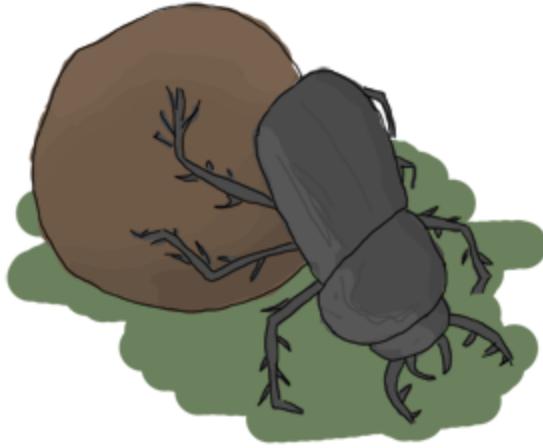
Specific Vs. Generic

What we've just done is get an understanding the core of OTP (conceptually speaking). This is what OTP really is all about: taking all the generic components, extracting them in libraries, making sure they work well and then reusing that code when possible. Then all that's left to do is focus on the specific stuff, things that will always change from application to application.

Obviously, there isn't much to save by doing things that way with only the kitty server. It looks a bit like abstraction for abstraction's sake. If the app we had to ship to a customer were nothing but the kitty server, then the first version might be fine. If you're going to have larger applications then it might be worth it to separate generic parts of your code from the specific sections.

Let's imagine for a moment that we have some Erlang software running on a server. Our software has a few kitty servers running, a veterinary process (you send your broken kitties and it returns them fixed), a kitty beauty salon, a server for pet food, supplies, etc. Most of these can be implemented with a client-server pattern. As time goes, your complex system becomes full of different servers running around.

Adding servers adds complexity in terms of code, but also in terms of testing, maintenance and understanding. Each implementation might be different, programmed in different styles by different people, and so on. However, if all these servers share the same common `my_server` abstraction, you substantially reduce that complexity. You understand the basic concept of the module instantly ("oh, it's a server!"), there's a single generic implementation of it to test, document, etc. The rest of the effort can be put on each specific implementation of it.



This means you reduce a lot of time tracking and solving bugs (just do it at one place for all servers). It also means that you reduce the number of bugs you introduce. If you were to re-write the `my_server:call/3` or the process' main loop all the time, not only would it be more time consuming, but chances of forgetting one step or the other would skyrocket, and so would bugs. Fewer bugs mean fewer calls during the night to go fix something, which is definitely good for all of us. Your mileage may vary, but I'll bet you don't appreciate going to the office on days off to fix bugs either.

Another interesting thing about what we did when separating the generic from the specific is that we instantly made it much easier to test our individual modules. If you wanted to unit test the old kitty server implementation, you'd need to spawn one process per test, give it the right state, send your messages and hope for the reply you expected. On the other hand, our second kitty server only requires us to run the function calls over the '`handle_call/3`' and '`handle_cast/2`' functions and see what they output as a new

state. No need to set up servers, manipulate the state. Just pass it in as a function parameter. Note that this also means the generic aspect of the server is much easier to test given you can just implement very simple functions that do nothing else than let you focus on the behaviour you want to observe, without the rest.

A much more 'hidden' advantage of using common abstractions in that way is that if everyone uses the exact same backend for their processes, when someone optimizes that single backend to make it a little bit faster, every process using it out there will run a little bit faster too. For this principle to work in practice, it's usually necessary to have a whole lot of people using the same abstractions and putting effort on them. Luckily for the Erlang community, that's what happens with the OTP framework.

Back to our modules. There are a bunch of things we haven't yet addressed: named processes, configuring the timeouts, adding debug information, what to do with unexpected messages, how to tie in hot code loading, handling specific errors, abstracting away the need to write most replies, handling most ways to shut a server down, making sure the server plays nice with supervisors, etc. Going over all of this is superfluous for this text, but would be necessary in real products that need to be shipped. Again, you might see why doing all of this by yourself is a bit of a risky task. Luckily for you (and the people who'll support your applications), the Erlang/OTP team managed to handle all of that for you with

the gen_server behaviour. gen_server is a bit like my_server on steroids, except it has years and years of testing and production use behind it.

Clients and Servers

Callback to the Future



The first OTP behaviour we'll see is one of the most used ones. Its name is `gen_server` and it has an interface a bit similar to the one we've written with `my_server` in [last chapter](#); it gives you a few functions to use it and in exchange, your module has to already have a few functions `gen_server` will use.

`init`

The first one is an `init/1` function. It is similar to the one we've used with `my_server` in that it is used to initialize the server's state and do all of these one-time tasks that it will depend on. The function can return `{ok, State}`, `{ok, State, TimeOut}`, `{ok, State, hibernate}`, `{stop, Reason}` or `ignore`.

The normal `{ok, State}` return value doesn't really need explaining past saying that `State` will be passed directly to the main loop of the process as the state to keep later on. The `TimeOut` variable is meant to be added to the tuple whenever you need a deadline before which you expect the server to receive a message. If no message is received before the deadline, a special one (the atom `timeout`) is sent to the server, which should be handled with `handle_info/2` (more on this later.)

On the other hand, if you do expect the process to take a long time before getting a reply and are worried about memory, you can add the `hibernate` atom to the tuple. Hibernation basically reduces the size of the process' state until it gets a message, at the cost of some processing power. If you are in doubt about using hibernation, you probably don't need it.

Returning {stop, Reason} should be done when something went wrong during the initialization.

Note: here's a more technical definition of process hibernation. It's no big deal if some readers do not understand or care about it. When the BIF erlang:hibernate(M,F,A) is called, the call stack for the currently running process is discarded (the function never returns). The garbage collection then kicks in, and what's left is one continuous heap that is shrunken to the size of the data in the process. This basically compacts all the data so the process takes less place.

Once the process receives a message, the function M:F with A as arguments is called and the execution resumes.

Note: while init/1 is running, execution is blocked in the process that spawned the server. This is because it is waiting for a 'ready' message sent automatically by the gen_server module to make sure everything went fine.

handle_call

The function handle_call/3 is used to work with synchronous messages (we'll see how to send them soon). It takes 3 arguments: *Request*, *From*, and *State*. It's pretty similar to how we programmed our own handle_call/3 in my_server. The biggest difference is how you reply to messages. In our own abstraction of a server, it was necessary to use my_server:reply/2 to talk back to the process. In the case of gen_servers, there are 8 different return values possible, taking the form of tuples.

Because there are many of them, here's a simple list instead:

```
{reply,Reply,NewState}  
{reply,Reply,NewState,Timeout}  
{reply,Reply,NewState,hibernate}  
{noreply,NewState}  
{noreply,NewState,Timeout}  
{noreply,NewState,hibernate}  
{stop,Reason,Reply,NewState}  
{stop,Reason,NewState}
```

For all of these, *Timeout* and *hibernate* work the same way as for init/1. Whatever is in *Reply* will be sent back to whoever called the server in the first place. Notice that there are three possible noreply options. When you use noreply, the generic part of the server will assume you're taking care of sending the reply back yourself. This

can be done with `gen_server:reply/2`, which can be used in the same way as `my_server:reply/2`.

Most of the time, you'll only need the reply tuples. There are still a few valid reasons to use `noreply`: whenever you want another process to send the reply for you or when you want to send an acknowledgement ('hey! I received the message!') but still process it afterwards (without replying this time), etc. If this is what you choose to do, it is absolutely necessary to use `gen_server:reply/2` because otherwise the call will time out and cause a crash.

handle_cast

The `handle_cast/2` callback works a lot like the one we had in `my_server`: it takes the parameters `Message` and `State` and is used to handle asynchronous calls. You do whatever you want in there, in a manner quite similar to what's doable with `handle_call/3`. On the other hand, only tuples without replies are valid return values:

```
{noreply,NewState}  
{noreply,NewState,Timeout}  
{noreply,NewState,hibernate}  
{stop,Reason,NewState}
```

handle_info

You know how I mentioned our own server didn't really deal with messages that do not fit our interface, right? Well `handle_info/2` is the solution. It's very similar to `handle_cast/2` and in fact returns the same tuples. The difference is that this callback is only there for messages that were sent directly with the `!` operator and special ones like `init/1`'s timeout, monitors' notifications and 'EXIT' signals.

terminate

The callback `terminate/2` is called whenever one of the three `handle_Something` functions returns a tuple of the form `{stop, Reason, NewState}` or `{stop, Reason, Reply, NewState}`. It takes two parameters, `Reason` and `State`, corresponding to the same values from the `stop` tuples.

`terminate/2` will also be called when its parent (the process that spawned it) dies, if and only if the `gen_server` is trapping exits.

Note: if any reason other than `normal`, `shutdown` or `{shutdown, Term}` is used when `terminate/2` is called, the OTP framework will see this as a failure and start logging a

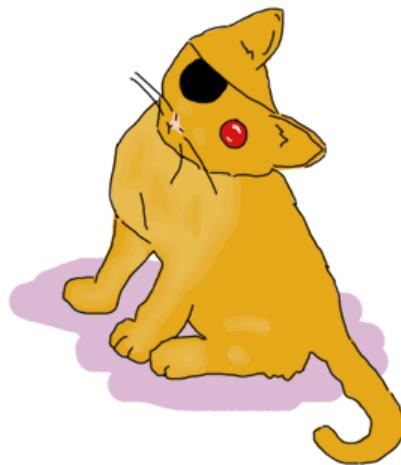
bunch of stuff here and there for you.

This function is pretty much the direct opposite of `init/1` so whatever was done in there should have its opposite in `terminate/2`. It's your server's janitor, the function in charge of locking the door after making sure everyone's gone. Of course, the function is helped by the VM itself, which should usually delete all ETS tables, close all ports, etc. for you. Note that the return value of this function doesn't really matter, because the code stops executing after it's been called.

code_change

The function `code_change/3` is there to let you upgrade code. It takes the form `code_change(PreviousVersion, State, Extra)`. Here, the variable `PreviousVersion` is either the version term itself in the case of an upgrade (read [More About Modules](#) again if you forget what this is), or `{down, Version}` in the case of a downgrade (just reloading older code). The `State` variable holds all of the current's server state so you can convert it.

Imagine for a moment that we used an orddict to store all of our data. However, as time goes on, the orddict becomes too slow and we decide to change it for a regular dict. In order to avoid the process crashing on the next function call, the conversion from one data structure to the other can be done in there, safely. All you have to do is return the new state with `{ok, NewState}`.



The `Extra` variable isn't something we'll worry about for now. It's mostly used in larger OTP deployment, where specific tools exist to upgrade entire releases on a VM. We're not there yet.

So now we've got all the callbacks defined. Don't worry if you're a bit lost: the OTP framework is a bit circular sometimes, where to understand part *A* of the framework you have to understand part *B*, but then part *B* requires to see part *A* to be useful. The best way to get over that confusion is to actually implement a `gen_server`.

.BEAM me up, Scotty!

This is going to be the `kitty_gen_server`. It's going to be mostly similar to `kitty_server2`, with only minimal API changes. First start a new module with the following lines in it:

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

And try to compile it. You should get something like this:

```
1> c(kitty_gen_server).  
.kitty_gen_server.erl:2: Warning: undefined callback function code_change/3 (behaviour 'gen_server')  
.kitty_gen_server.erl:2: Warning: undefined callback function handle_call/3 (behaviour 'gen_server')  
.kitty_gen_server.erl:2: Warning: undefined callback function handle_cast/2 (behaviour 'gen_server')  
.kitty_gen_server.erl:2: Warning: undefined callback function handle_info/2 (behaviour 'gen_server')  
.kitty_gen_server.erl:2: Warning: undefined callback function init/1 (behaviour 'gen_server')  
.kitty_gen_server.erl:2: Warning: undefined callback function terminate/2 (behaviour 'gen_server')  
{ok,kitty_gen_server}
```

The compilation worked, but there are warnings about missing callbacks. This is because of the `gen_server` behaviour. A behaviour is basically a way for a module to specify functions it expects another module to have. The behaviour is the contract sealing the deal between the well-behaved generic part of the code and the specific, error-prone part of the code (yours).

Note: both 'behavior' and 'behaviour' are accepted by the Erlang compiler.

Defining your own behaviours is really simple. You just need to export a function called `behaviour_info/1` implemented as follows:

```
-module(my_behaviour).  
-export([behaviour_info/1]).
```

```
%% init/1, some_fun/0 and other/3 are now expected callbacks  
behaviour_info(callbacks) -> [{init,1}, {some_fun, 0}, {other, 3}];  
behaviour_info(_) -> undefined.
```

And that's about it for behaviours. You can just use `-behaviour(my_behaviour)`, in a module implementing them to get compiler warnings if you forgot a function. Anyway, back to our third kitty server.

The first function we had was `start_link/0`. This one can be changed to the following:

```
start_link() -> gen_server:start_link(?MODULE, [], []).
```

The first parameter is the callback module, the second one is the list of parameters to pass to `init/1` and the third one is about debugging options that won't be covered right now. You could add a fourth parameter in the first position, which would be the name to register the server with. Note that while the previous version of the function simply returned a pid, this one instead returns `{ok, Pid}`.

Next functions now:

```
%% Synchronous call
order_cat(Pid, Name, Color, Description) ->
    gen_server:call(Pid, {order, Name, Color, Description}).
```

```
%% This call is asynchronous
return_cat(Pid, Cat = #cat{}) ->
    gen_server:cast(Pid, {return, Cat}).
```

```
%% Synchronous call
close_shop(Pid) ->
    gen_server:call(Pid, terminate).
```

All of these calls are a one-to-one change. Note that a third parameter can be passed to `gen_server:call/2-3` to give a timeout. If you don't give a timeout to the function (or the atom `infinity`), the default is set to 5 seconds. If no reply is received before time is up, the call crashes.

Now we'll be able to add the `gen_server` callbacks. The following table shows the relationship we have between calls and callbacks:

gen_server YourModule	
start/3-4	init/1
start_link/3-4	init/1
call/2-3	handle_call/3
cast/2	handle_cast/2

And then you have the other callbacks, those that are more about special cases:

- handle_info/2
- terminate/2
- code_change/3

Let's begin by changing those we already have to fit the model: init/1, handle_call/3 and handle_cast/2.

```
%%% Server functions
init([]) -> {ok, []}. %% no treatment of info here!
```

```
handle_call({order, Name, Color, Description}, _From, Cats) ->
  if Cats =:= [] ->
    {reply, make_cat(Name, Color, Description), Cats};
  Cats =/= [] ->
    {reply, hd(Cats), tl(Cats)}
  end;
handle_call(terminate, _From, Cats) ->
  {stop, normal, ok, Cats}.
```

```
handle_cast({return, Cat = #cat{}}, Cats) ->
  {noreply, [Cat|Cats]}.
```

Again, very little has changed there. In fact, the code is now shorter, thanks to smarter abstractions. Now we get to the new callbacks. The first one is handle_info/2. Given this is a toy module and we have no logging system predefined, just outputting the unexpected messages will be enough:

```
handle_info(Msg, Cats) ->
  io:format("Unexpected message: ~p~n",[Msg]),
  {noreply, Cats}.
```

The next one is the terminate/2 callback. It will be very similar to the terminate/1 private function we had:

```
terminate(normal, Cats) ->
  [io:format("~p was set free.~n",[C#cat.name]) || C <- Cats],
  ok.
```

And then the last callback, code_change/3:

```
code_change(_OldVsn, State, _Extra) ->
  %% No change planned. The function is there for the behaviour,
```

```
%% but will not be used. Only a version on the next
{ok, State}.
```

Just remember to keep in the `make_cat/3` private function:

```
%%% Private functions
make_cat(Name, Col, Desc) ->
    #cat{name=Name, color=Col, description=Desc}.
```

And we can now try the brand new code:

```
1> c(kitty_gen_server).
{ok,kitty_gen_server}
2> rr(kitty_gen_server).
[cat]
3> {ok, Pid} = kitty_gen_server:start_link().
{ok,<0.253.0>}
4> Pid ! <<"Test handle_info">>.
Unexpected message: <<"Test handle_info">>
<<"Test handle_info">>
5> Cat = kitty_gen_server:order_cat(Pid, "Cat Stevens", white, "not actually a cat").
#cat{name = "Cat Stevens",color = white,
     description = "not actually a cat"}
6> kitty_gen_server:return_cat(Pid, Cat).
ok
7> kitty_gen_server:order_cat(Pid, "Kitten Mittens", black, "look at them little paws!").
#cat{name = "Cat Stevens",color = white,
     description = "not actually a cat"}
8> kitty_gen_server:order_cat(Pid, "Kitten Mittens", black, "look at them little paws!").
#cat{name = "Kitten Mittens",color = black,
     description = "look at them little paws!"}
9> kitty_gen_server:return_cat(Pid, Cat).
ok
10> kitty_gen_server:close_shop(Pid).
"Cat Stevens" was set free.
ok
```



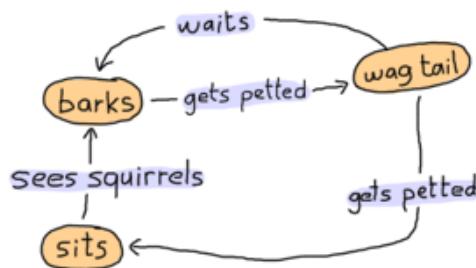
Oh and hot damn, it works!

So what can we say about this generic adventure? Probably the same generic stuff as before: separating the generic from the specific is a great idea on every point. Maintenance is simpler, complexity is reduced, the code is safer, easier to test and less prone to bugs. If there are bugs, they are easier to fix. Generic servers are only one of the many available abstractions, but they're certainly one of the most used ones. We'll see more of these abstractions and behaviours in the next chapters.

Rage Against The Finite-State Machines

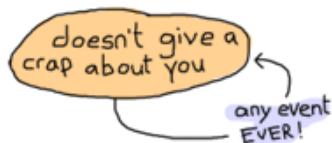
What Are They?

A finite-state machine (FSM) is not really a machine, but it does have a finite number of states. I've always found finite-state machines easier to understand with graphs and diagrams. For example, the following would be a simplistic diagram for a (very dumb) dog as a state machine:



Here the dog has 3 states: sitting, barking or wagging its tail. Different events or inputs may force it to change its state. If a dog is calmly sitting and sees a squirrel, it will start barking and won't stop until you pet it again. However, if the dog is sitting and you pet it, we have no idea what might happen. In the Erlang world, the dog could crash (and eventually be restarted by its supervisor). In the real world that would be a freaky event, but your dog would come back after being ran over by a car, so it's not all bad.

Here's a cat's state diagram for a comparison:



This cat has a single state, and no event can ever change it.

Implementing the cat state machine in Erlang is a fun and simple task:

```
-module(cat_fsm).
-export([start/0, event/2]).

start() ->
    spawn(fun() -> dont_give_crap() end).

event(Pid, Event) ->
    Ref = make_ref(), % won't care for monitors here
    Pid ! {self(), Ref, Event},
    receive
        {Ref, Msg} -> {ok, Msg}
    after 5000 ->
        {error, timeout}
    end.

dont_give_crap() ->
    receive
        {Pid, Ref, _Msg} -> Pid ! {Ref, meh};
        _ -> ok
    end,
    io:format("Switching to 'dont_give_crap' state~n"),
    dont_give_crap().
```

We can try the module to see that the cat really never gives a crap:

```
1> c(cat_fsm).
{ok,cat_fsm}
2> Cat = cat_fsm:start().
<0.670>
3> cat_fsm:event(Cat, pet).
Switching to 'dont_give_crap' state
{ok,meh}
4> cat_fsm:event(Cat, love).
Switching to 'dont_give_crap' state
{ok,meh}
5> cat_fsm:event(Cat, cherish).
```

```
Switching to 'dont_give_crap' state
{ok,meh}
```

The same can be done for the dog FSM, except more states are available:

```
-module(dog_fsm).
-export([start/0, squirrel/1, pet/1]).
```

```
start() ->
    spawn(fun() -> bark() end).
```

```
squirrel(Pid) -> Pid ! squirrel.
```

```
pet(Pid) -> Pid ! pet.
```

```
bark() ->
    io:format("Dog says: BARK! BARK!~n"),
    receive
        pet ->
            wag_tail();
        _ ->
            io:format("Dog is confused~n"),
            bark()
    after 2000 ->
        bark()
    end.
```

```
wag_tail() ->
    io:format("Dog wags its tail~n"),
    receive
        pet ->
            sit();
        _ ->
            io:format("Dog is confused~n"),
            wag_tail()
    after 30000 ->
        bark()
    end.
```

```

sit() ->
    io:format("Dog is sitting. Gooooood boy!~n"),
    receive
        squirrel ->
            bark();
        _ ->
            io:format("Dog is confused~n"),
            sit()
    end.

```

It should be relatively simple to match each of the states and transitions to what was on the diagram above. Here's the FSM in use:

```

6> c(dog_fsm).
{ok,dog_fsm}
7> Pid = dog_fsm:start().
Dog says: BARK! BARK!
<0.46.0>
Dog says: BARK! BARK!
Dog says: BARK! BARK!
Dog says: BARK! BARK!
8> dog_fsm:pet(Pid).
pet
Dog wags its tail
9> dog_fsm:pet(Pid).
Dog is sitting. Gooooood boy!
pet
10> dog_fsm:pet(Pid).
Dog is confused
pet
Dog is sitting. Gooooood boy!
11> dog_fsm:squirrel(Pid).
Dog says: BARK! BARK!
squirrel
Dog says: BARK! BARK!
12> dog_fsm:pet(Pid).
Dog wags its tail
pet

```

```
13> %% wait 30 seconds
Dog says: BARK! BARK!
Dog says: BARK! BARK!
Dog says: BARK! BARK!
13> dog_fsm:pet(Pid).
Dog wags its tail
pet
14> dog_fsm:pet(Pid).
Dog is sitting. Gooooood boy!
pet
```

You can follow along with the schema if you want (I usually do, it helps being sure that nothing's wrong).

That's really the core of FSMs implemented as Erlang processes. There are things that could have been done differently: we could have passed state in the arguments of the state functions in a way similar to what we do with servers' main loop. We could also have added an init and terminate functions, handled code updates, etc.

Another difference between the dog and cat FSMs is that the cat's events are *synchronous* and the dog's events are *asynchronous*. In a real FSM, both could be used in a mixed manner, but I went for the simplest representation out of pure untapped laziness. There are other forms of event the examples do not show: global events that can happen in any state.

One example of such an event could be when the dog gets a sniff of food. Once the smell food event is triggered, no matter what state the dog is in, he'd go looking for the source of food.

Now we won't spend too much time implementing all of this in our 'written-on-a-napkin' FSM. Instead we'll move directly to the gen_fsm behaviour.

Generic Finite-State Machines

The `gen_fsm` behaviour is somewhat similar to `gen_server` in that it is a specialised version of it. The biggest difference is that rather than handling *calls* and *casts*, we're handling *synchronous* and *asynchronous* events. Much like our dog and cat examples, each state is represented by a function. Again, we'll go through the callbacks our modules need to implement in order to work.

init

This is the same `init/1` as used for generic servers, except the return values accepted are `{ok, StateName, Data}`, `{ok, StateName, Data, Timeout}`, `{ok, StateName, Data, hibernate}` and `{stop, Reason}`. The `stop` tuple works in the same manner as for `gen_servers`, and `hibernate` and `Timeout` keep the same semantics.

What's new here is that `StateName` variable. `StateName` is an atom and represents the next callback function to be called.



StateName

The functions `StateName/2` and `StateName/3` are placeholder names and you are to decide what they will be. Let's suppose the `init/1` function returns the tuple `{ok, sitting, dog}`. This means the finite state machine will be in a `sitting` state. This is not the same kind of state as we had seen with `gen_server`; it is rather equivalent to the `sit`, `bark` and

wag_tail states of the previous dog FSM. These states dictate a context in which you handle a given event.

An example of this would be someone calling you on your phone. If you're in the state 'sleeping on a Saturday morning', your reaction might be to yell in the phone. If your state is 'waiting for a job interview', chances are you'll pick the phone and answer politely. On the other hand, if you're in the state 'dead', then I am surprised you can even read this text at all.

Back to our FSM. The init/1 function said we should be in the sitting state. Whenever the gen_fsm process receives an event, either the function sitting/2 or sitting/3 will be called. The sitting/2 function is called for asynchronous events and sitting/3 for synchronous ones.

The arguments for sitting/2 (or generally StateName/2) are *Event*, the actual message sent as an event, and *StateData*, the data that was carried over the calls. sitting/2 can then return the tuples {next_state, NextStateName, NewStateData}, {next_state, NextStateName, NewStateData, Timeout}, {next_state, NextStateName, NewStateData, hibernate} and {stop, Reason, NewStateData}.

The arguments for sitting/3 are similar, except there is a *From* variable in between *Event* and *StateData*. The *From* variable is used in exactly the same way as it was for gen_servers, including gen_fsm:reply/2. The StateName/3 functions can return the following tuples:

```
{reply, Reply, NextStateName, NewStateData}  
{reply, Reply, NextStateName, NewStateData, Timeout}  
{reply, Reply, NextStateName, NewStateData, hibernate}  
  
{next_state, NextStateName, NewStateData}  
{next_state, NextStateName, NewStateData, Timeout}  
{next_state, NextStateName, NewStateData, hibernate}
```

```
{stop, Reason, Reply, NewStateData}  
{stop, Reason, NewStateData}
```

Note that there's no limit on how many of these functions you can have, as long as they are exported. The atoms returned as *NextStateName* in the tuples will determine whether the function will be called or not.

handle_event

In the last section, I mentioned global events that would trigger a specific reaction no matter what state we're in (the dog smelling food will drop whatever it is doing and will instead look for food). For these events that should be treated the same way in every state, the `handle_event/3` callback is what you want. The function takes arguments similar to `StateName/2` with the exception that it accepts a `StateName` variable in between them, telling you what the state was when the event was received. It returns the same values as `StateName/2`.

handle_sync_event

The `handle_sync_event/4` callback is to `StateName/3` what `handle_event/2` is to `StateName/2`. It handles synchronous global events, takes the same parameters and returns the same kind of tuples as `StateName/3`.

Now might be a good time to explain how we know whether an event is global or if it's meant to be sent to a specific state. To determine this we can look at the function used to send an event to the FSM. Asynchronous events aimed at any `StateName/2` function are sent with `send_event/2`, synchronous events to be picked up by `StateName/3` are to be sent with `sync_send_event/2-3`.

The two equivalent functions for global events are `send_all_state_event/2` and `sync_send_all_state_event/2-3` (quite a long name).

code_change

This works exactly the same as it did for `gen_servers` except that it takes an extra state parameter when called like `code_change(OldVersion, StateName, Data, Extra)`, and returns a tuple of the form `{ok, NextStateName, NewStateData}`.

terminate

This should, again, act a bit like what we have for generic servers. `terminate/3` should do the opposite of `init/1`.

A Trading System Specification

It's time to put all of this in practice. Many Erlang tutorials about finite-state machines use examples containing telephone switches and similar things. It's my guess that most programmers will rarely have to deal with telephone switches for state machines. Because of that, we're going to look at an example which is more fitting for many developers: we'll design and implement an item trading system for some fictional and non-existing video game.

The design I have picked is somewhat challenging. Rather than using a broker through which players route items and confirmations (which, frankly, would be easier), we're going to implement a server where both players speak to each other directly (which would have the advantage of being distributable).

Because the implementation is tricky, I'll spend a good while describing it, the kind of problems to be faced and the ways to fix them.

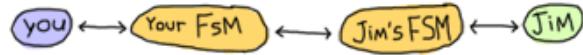
First of all, we should define the actions that can be done by our players when trading. The first is asking for a trade to be set up. The other user should also be able to accept that trade. We won't give them the right to deny a trade, though, because we want to keep things simple. It will be easy to add this feature once the whole thing is done.

Once the trade is set up, our users should be able to negotiate with each other. This means they should be able to make offers and then retract them if they want. When both players are satisfied with the offer, they can each declare themselves as ready to finalise the trade. The data should then be saved somewhere on both sides. At any point in time, it should also make sense for any of the players to cancel the whole trade. Some *pleb* could be offering only items deemed unworthy to the other party (who might be very busy) and so it should be possible to backhand them with a well-deserved cancellation.

In short, the following actions should be possible:

- ask for a trade
- accept a trade
- offer items
- retract an offer
- declare self as ready
- brutally cancel the trade

Now, when each of these actions is taken, the other player's FSM should be made aware of it. This makes sense, because when Jim tells his FSM to send an item to Carl, Carl's FSM has to be made aware of it. This means both players can talk to their own FSM, which will talk to the other's FSM. This gives us something a bit like this:



The first thing to notice when we have two identical processes communicating with each other is that we have to avoid synchronous calls as much as possible. The reason for this is that if Jim's FSM sends a message to Carl's FSM and then waits for its reply while at the same time Carl's FSM sends a message over to Jim's FSM and waits for its own specific reply, both end up waiting for the other without ever replying. This effectively freezes both FSMs. We have a deadlock.

One solution to this is to wait for a timeout and then move on, but then there will be leftover messages in both processes' mailboxes and the protocol will be messed up. This certainly is a can of worms, and so we want to avoid it.

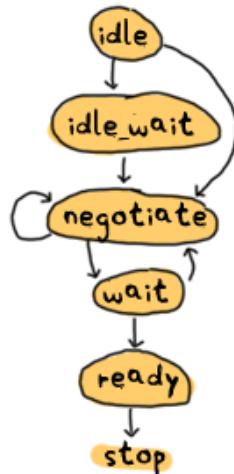
The simplest way to do it is to avoid all synchronous messages and go fully asynchronous. Note that Jim might still make a synchronous call to his own FSM; there's no risk here because the FSM won't need to call Jim and so no deadlock can occur between them.

When two of these FSMs communicate together, the whole exchange might look a bit like this:

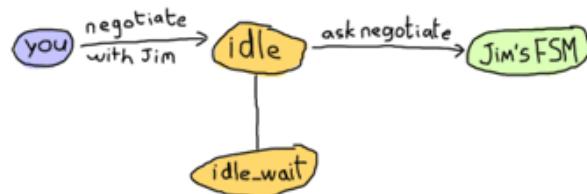


Both FSMs are in an idle state. When you ask Jim to trade, Jim has to accept before things move on. Then both of you can offer items or withdraw them. When you are both declaring yourself ready, the trade can take place. This is a simplified version of all that can happen and we'll see all possible cases with more detail in the next paragraphs.

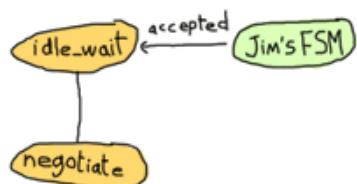
Here comes the tough part: defining the state diagram and how state transitions happen. Usually a good bit of thinking goes into this, because you have to think of all the small things that could go wrong. Some things might go wrong even after having reviewed it many times. Because of this, I'll simply put the one I decided to implement here and then explain it.



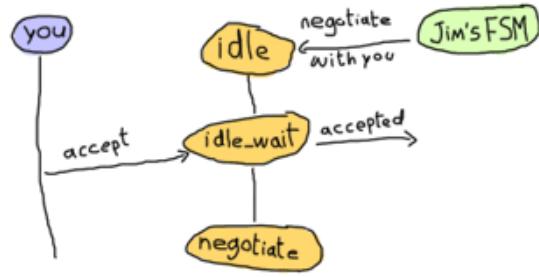
At first, both finite-state machines start in the `idle` state. At this point, one thing we can do is ask some other player to negotiate with us:



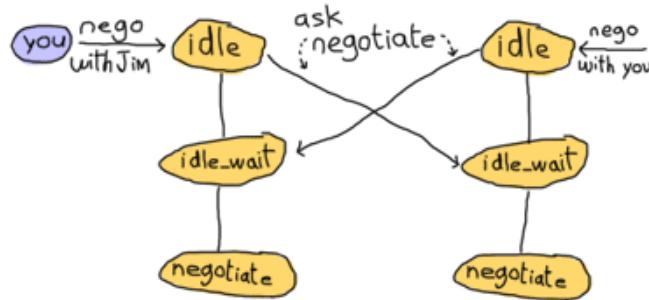
We go into `idle_wait` mode in order to wait for an eventual reply after our FSM forwarded the demand. Once the other FSM sends the reply, ours can switch to `negotiate`:



The other player should also be in `negotiate` state after this. Obviously, if we can invite the other, the other can invite us. If all goes well, this should end up looking like this:

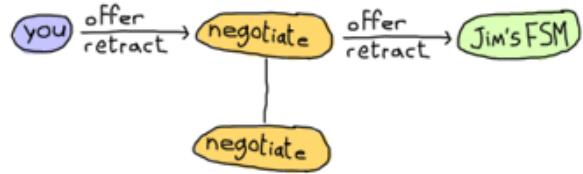


So this is pretty much the opposite as the two previous state diagrams bundled into one. Note that we expect the player to accept the offer in this case. What happens if by pure luck, we ask the other player to trade with us at the same time he asks us to trade?

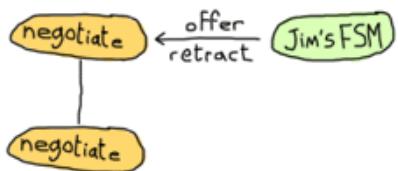


What happens here is that both clients ask their own FSM to negotiate with the other one. As soon as the *ask negotiate* messages are sent, both FSMs switch to *idle_wait* state. Then they will be able to process the negotiation question. If we review the previous state diagrams, we see that this combination of events is the only time we'll receive *ask negotiate* messages while in the *idle_wait* state. Consequently, we know that getting these messages in *idle_wait* means that we hit the race condition and can assume both users want to talk to each other. We can move both of them to *negotiate* state. Hooray.

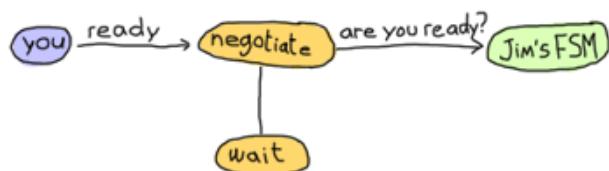
So now we're negotiating. According to the list of actions I listed earlier, we must support users offering items and then retracting the offer:



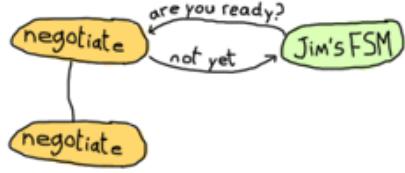
All this does is forward our client's message to the other FSM. Both finite-state machines will need to hold a list of items offered by either player, so they can update that list when receiving such messages. We stay in the `negotiate` state after this; maybe the other player wants to offer items too:



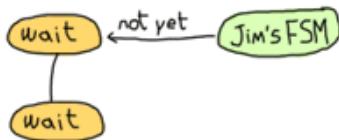
Here, our FSM basically acts in a similar manner. This is normal. Once we get tired of offering things and think we're generous enough, we have to say we're ready to officialise the trade. Because we have to synchronise both players, we'll have to use an intermediary state, as we did for `idle` and `idle_wait`:



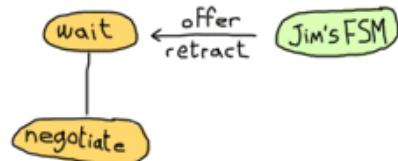
What we do here is that as soon as our player is ready, our FSM asks Jim's FSM if he's ready. Pending its reply, our own FSM falls into its `wait` state. The reply we'll get will depend on Jim's FSM state: if it's in `wait` state, it'll tell us that it's ready. Otherwise, it'll tell us that it's not ready yet. That's precisely what our FSM automatically replies to Jim if he asks us if we are ready when in `negotiate` state:



Our finite state machine will remain in `negotiate` mode until our player says he's ready. Let's assume he did and we're now in the `wait` state. However, Jim's not there yet. This means that when we declared ourselves as ready, we'll have asked Jim if he was also ready and his FSM will have replied 'not yet':



He's not ready, but we are. We can't do much but keep waiting. While waiting after Jim, who's still negotiating by the way, it is possible that he will try to send us more items or maybe cancel his previous offers:

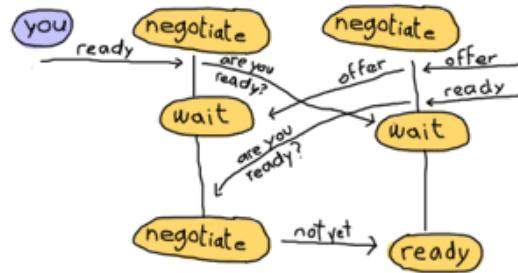


Of course, we want to avoid Jim removing all of his items and then clicking "I'm ready!", screwing us over in the process. As soon as he changes the items offered, we go back into the `negotiate` state so we can either modify our own offer, or examine the current one and decide we're ready. Rinse and repeat.

At some point, Jim will be ready to finalise the trade too. When this happens, his finite-state machine will ask ours if we are ready:

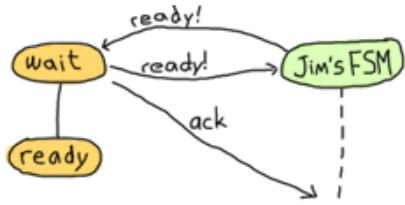


What our FSM does is reply that we indeed are ready. We stay in the waiting state and refuse to move to the ready state though. Why is this? Because there's a potential race condition! Imagine that the following sequence of events takes place, without doing this necessary step:



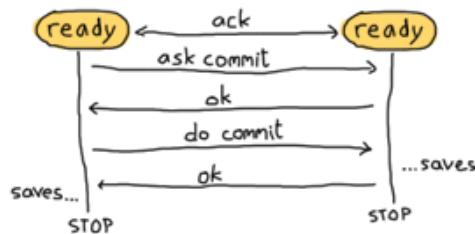
This is a bit complex, so I'll explain. Because of the way messages are received, we could possibly only process the item offer *after* we declared ourselves ready and also *after* Jim declared himself as ready. This means that as soon as we read the offer message, we switch back to negotiate state. During that time, Jim will have told us he is ready. If he were to change states right there and move on to ready (as illustrated above), he'd be caught waiting indefinitely while we wouldn't know what the hell to do. This could also happen the other way around! Ugh.

One way to solve this is by adding one layer of indirection (Thanks to David Wheeler). This is why we stay in wait mode and send 'ready!' (as shown in our previous state diagram). Here's how we deal with that 'ready!' message, assuming we were already in the ready state because we told our FSM we were ready beforehand:



When we receive 'ready!' from the other FSM, we send 'ready!' back again. This is to make sure that we won't have the 'double race condition' mentioned above. This will create a superfluous 'ready!' message in one of the two FSMs, but we'll just have to ignore it in this case. We then send an 'ack' message (and the Jim's FSM will do the same) before moving to ready state. The reason why this 'ack' message exists is due to some implementation details about synchronising clients. I've put it in the diagram for the sake of being correct, but I won't explain it until later. Forget about it for now. We finally managed to synchronise both players. Whew.

So now there's the ready state. This one is a bit special. Both players are ready and have basically given the finite-state machines all the control they need. This lets us implement a bastardized version of a two-phase commit to make sure things go right when making the trade official:



Our version (as described above) will be rather simplistic. Writing a truly correct two-phase commit would require a lot more code than what is necessary for us to understand finite-state machines.

Finally, we only have to allow the trade to be cancelled at any time. This means that somehow, no matter what state we're in, we're going to listen to the 'cancel' message from both sides and quit the transaction. It should also be common courtesy to let the other side know we're gone before leaving.

Alright! It's a whole lot of information to absorb at once. Don't worry if it takes a while to fully grasp it. It took a bunch of people to look over my protocol to see if it was right, and even then we all missed a few race conditions that I then caught a few days later when reviewing the code while writing this text. It's normal to need to read it more than once, especially if you are not used to asynchronous protocols. If this is the case, I fully encourage you to try and design your own protocol. Then ask yourself "what happens if two people do the same actions very fast? What if they chain two other events quickly? What do I do with messages I don't handle when changing states?" You'll see that the complexity grows real fast. You might find a solution similar to mine, possibly a better one (let me know if this is the case!) No matter the outcome, it's a very interesting thing to work on and our FSMs are still relatively simple.

Once you've digested all of this (or before, if you're a rebel reader), you can go to the next section, where we implement the gaming system. For now you can take a nice coffee break if you feel like doing so.



Game trading between two players

The first thing that needs to be done to implement our protocol with OTP's gen_fsm is to create the interface. There will be 3 callers for our module: the player, the gen_fsm behaviour and the other player's FSM. We will only need to export the player function and gen_fsm functions, though. This is because the other FSM will also run within the trade_fsm module and can access them from the inside:

```
-module(trade_fsm).
-behaviour(gen_fsm).

%% public API
-export([start/1, start_link/1, trade/2, accept_trade/1,
        make_offer/2, retract_offer/2, ready/1, cancel/1]).

%% gen_fsm callbacks
-export([init/1, handle_event/3, handle_sync_event/4, handle_info/3,
        terminate/3, code_change/4,
        % custom state names
        idle/2, idle/3, idle_wait/2, idle_wait/3, negotiate/2,
        negotiate/3, wait/2, ready/2, ready/3]).
```

So that's our API. You can see I'm planning on having some functions being both synchronous and asynchronous. This is mostly because we want our client to call us synchronously in some cases, but the other FSM can do it asynchronously. Having the client synchronous simplifies our logic a whole lot by limiting the number of contradicting messages that can be sent one after the other. We'll get there. Let's first implement the actual public API according to the protocol defined above:

```
%%% PUBLIC API
start(Name) ->
    gen_fsm:start(?MODULE, [Name], []).

start_link(Name) ->
    gen_fsm:start_link(?MODULE, [Name], []).
```

```
%% ask for a begin session. Returns when/if the other accepts
trade(OwnPid, OtherPid) ->
    gen_fsm:sync_send_event(OwnPid, {negotiate, OtherPid}, 30000).
```

```
%% Accept someone's trade offer.
accept_trade(OwnPid) ->
    gen_fsm:sync_send_event(OwnPid, accept_negotiate).
```

```
%% Send an item on the table to be traded
make_offer(OwnPid, Item) ->
    gen_fsm:send_event(OwnPid, {make_offer, Item}).
```

```
%% Cancel trade offer
retract_offer(OwnPid, Item) ->
    gen_fsm:send_event(OwnPid, {retract_offer, Item}).
```

```
%% Mention that you're ready for a trade. When the other
%% player also declares being ready, the trade is done
ready(OwnPid) ->
    gen_fsm:sync_send_event(OwnPid, ready, infinity).
```

```
%% Cancel the transaction.
cancel(OwnPid) ->
    gen_fsm:sync_send_all_state_event(OwnPid, cancel).
```

This is rather standard; all these 'gen_fsm' functions have been covered before (except start/3-4 and start_link/3-4 which I believe you can figure out) in this chapter.

Next we'll implement the FSM to FSM functions. The first ones have to do with trade setups, when we first want to ask the other user to join us in a trade:

```
%% Ask the other FSM's Pid for a trade session
ask_negotiate(OtherPid, OwnPid) ->
    gen_fsm:send_event(OtherPid, {ask_negotiate, OwnPid}).
```

```
%% Forward the client message accepting the transaction
accept_negotiate(OtherPid, OwnPid) ->
    gen_fsm:send_event(OtherPid, {accept_negotiate, OwnPid}).
```

The first function asks the other pid if they want to trade, and the second one is used to reply to it (asynchronously, of course).

We can then write the functions to offer and cancel offers. According to our protocol above, this is what they should be like:

```
%% forward a client's offer
do_offer(OtherPid, Item) ->
    gen_fsm:send_event(OtherPid, {do_offer, Item}).
```

```
%% forward a client's offer cancellation
undo_offer(OtherPid, Item) ->
    gen_fsm:send_event(OtherPid, {undo_offer, Item}).
```

So, now that we've got these calls done, we need to focus on the rest. The remaining calls relate to being ready or not and handling the final commit. Again, given our protocol above, we have three calls: are_you_ready, which can have the replies not_yet or ready!:

```
%% Ask the other side if he's ready to trade.
are_you_ready(OtherPid) ->
    gen_fsm:send_event(OtherPid, are_you_ready).
```

```
%% Reply that the side is not ready to trade
%% i.e. is not in 'wait' state.
not_yet(OtherPid) ->
    gen_fsm:send_event(OtherPid, not_yet).
```

```
%% Tells the other fsm that the user is currently waiting
%% for the ready state. State should transition to 'ready'
am_ready(OtherPid) ->
    gen_fsm:send_event(OtherPid, 'ready!').
```

The only functions left are those which are to be used by both FSMs when doing the commit in the ready state. Their precise usage will be described more in detail later, but for now, the names and the sequence/state diagram from earlier should be enough. Nonetheless, you can still transcribe them to your own version of trade_fsm:

```
%% Acknowledge that the fsm is in a ready state.  
ack_trans(OtherPid) ->  
    gen_fsm:send_event(OtherPid, ack).  
  
%% ask if ready to commit  
ask_commit(OtherPid) ->  
    gen_fsm:sync_send_event(OtherPid, ask_commit).  
  
%% begin the synchronous commit  
do_commit(OtherPid) ->  
    gen_fsm:sync_send_event(OtherPid, do_commit).
```

Oh and there's also the courtesy function allowing us to warn the other FSM we cancelled the trade:

```
notify_cancel(OtherPid) ->  
    gen_fsm:send_all_state_event(OtherPid, cancel).
```

We can now move to the really interesting part: the gen_fsm callbacks. The first callback is init/1. In our case, we'll want each FSM to hold a name for the user it represents (that way, our output will be nicer) in the data it keeps passing on to itself. What else do we want to hold in memory? In our case, we want the other's pid, the items we offer and the items the other offers. We're also going to add the reference of a monitor (so we know to abort if the other dies) and a from field, used to do delayed replies:

```
-record(state, {name = "",  
               other,  
               ownitems = [],  
               otheritems = []},
```

```
monitor,  
from}).
```

In the case of init/1, we'll only care about our name for now. Note that we'll begin in the idle state:

```
init(Name) ->  
{ok, idle, #state{name=Name}}.
```

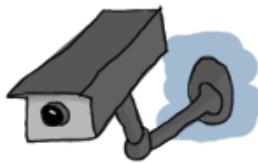
The next callbacks to consider would be the states themselves. So far I've described the state transitions and calls that can be made, but We'll need a way to make sure everything goes alright. We'll write a few utility functions first:

```
%% Send players a notice. This could be messages to their clients  
%% but for our purposes, outputting to the shell is enough.  
notice(#state{name=N}, Str, Args) ->  
    io:format("~s: ++Str++~n", [N|Args]).
```

```
%% Unexpected allows to log unexpected messages  
unexpected(Msg, State) ->  
    io:format("~p received unknown event ~p while in state ~p~n",  
        [self(), Msg, State]).
```

And we can start with the idle state. For the sake of convention, I'll cover the asynchronous version first. This one shouldn't need to care for anything but the other player asking for a trade given our own player, if you look at the API functions, will use a synchronous call:

```
idle({ask_negotiate, OtherPid}, S=#state{}) ->  
    Ref = monitor(process, OtherPid),  
    notice(S, "~p asked for a trade negotiation", [OtherPid]),  
    {next_state, idle_wait, S#state{other=OtherPid, monitor=Ref}};  
idle(Event, Data) ->  
    unexpected(Event, idle),  
    {next_state, idle, Data}.
```



A monitor is set up to allow us to handle the other dying, and its ref is stored in the FSM's data along with the other's pid, before moving to the idle_wait state. Note that we will report all unexpected events and ignore them by staying in the state we were already in. We can have a few out of band messages here and there that could be the result of race conditions. It's usually safe to ignore them, but we can't easily get rid of them. It's just better not to crash the whole FSM on these unknown, but somewhat expected messages.

When our own client asks the FSM to contact another player for a trade, it will send a synchronous event. The idle/3 callback will be needed:

```
idle({negotiate, OtherPid}, From, S=#state{}) ->
    ask_negotiate(OtherPid, self()),
    notice(S, "asking user ~p for a trade", [OtherPid]),
    Ref = monitor(process, OtherPid),
    {next_state, idle_wait, S#state{other=OtherPid, monitor=Ref, from=From}};
idle(Event, _From, Data) ->
    unexpected(Event, idle),
    {next_state, idle, Data}.
```

We proceed in a way similar to the asynchronous version, except we need to actually ask the other side whether they want to negotiate with us or not. You'll notice that we do *not* reply to the client yet. This is because we have nothing interesting to say, and we want the client locked and waiting for the trade to be accepted before doing anything. The reply will only be sent if the other side accepts once we're in idle_wait.

When we're there, we have to deal with the other accepting to negotiate and the other asking to negotiate (the result of a race condition, as described in the protocol):

```
idle_wait({ask_negotiate, OtherPid}, S=#state{other=OtherPid}) ->
    gen_fsm:reply(S#state.from, ok),
    notice(S, "starting negotiation", []),
    {next_state, negotiate, S};

%% The other side has accepted our offer. Move to negotiate state
idle_wait({accept_negotiate, OtherPid}, S=#state{other=OtherPid}) ->
    gen_fsm:reply(S#state.from, ok),
    notice(S, "starting negotiation", []),
    {next_state, negotiate, S};

idle_wait(Event, Data) ->
    unexpected(Event, idle_wait),
    {next_state, idle_wait, Data}.
```

This gives us two transitions to the `negotiate` state, but remember that we must use `gen_fsm:reply/2` reply to our client to tell it it's okay to start offering items. There's also the case of our FSM's client accepting the trade suggested by the other party:

```
idle_wait(accept_negotiate, _From, S=#state{other=OtherPid}) ->
    accept_negotiate(OtherPid, self()),
    notice(S, "accepting negotiation", []),
    {reply, ok, negotiate, S};

idle_wait(Event, _From, Data) ->
    unexpected(Event, idle_wait),
    {next_state, idle_wait, Data}.
```

Again, this one moves on to the `negotiate` state. Here, we must handle asynchronous queries to add and remove items coming both from the client and the other FSM. However, we have not yet decided how to store items. Because I'm somewhat lazy and I assume users won't trade that many items, simple lists will do it for now. However, we might change our mind at a later point, so it would be a good idea to

wrap item operations in their own functions. Add the following functions at the bottom of the file with notice/3 and unexpected/2:

```
%% adds an item to an item list
```

```
add(Item, Items) ->
```

```
    [Item | Items].
```

```
%% remove an item from an item list
```

```
remove(Item, Items) ->
```

```
    Items -- [Item].
```

Simple, but they have the role of isolating the actions (adding and removing items) from their implementation (using lists). We could easily move to prolists, arrays or whatever data structure without disrupting the rest of the code.

Using both of these functions, we can implement offering and removing items:

```
negotiate({make_offer, Item}, S=#state{ownitems=OwnItems}) ->
    do_offer(S#state.other, Item),
    notice(S, "offering ~p", [Item]),
    {next_state, negotiate, S#state{ownitems=add(Item, OwnItems)}};

%% Own side retracting an item offer
negotiate({retract_offer, Item}, S=#state{ownitems=OwnItems}) ->
    undo_offer(S#state.other, Item),
    notice(S, "cancelling offer on ~p", [Item]),
    {next_state, negotiate, S#state{ownitems=remove(Item, OwnItems)}};

%% other side offering an item
negotiate({do_offer, Item}, S=#state{otheritems=OtherItems}) ->
    notice(S, "other player offering ~p", [Item]),
    {next_state, negotiate, S#state{otheritems=add(Item, OtherItems)}};

%% other side retracting an item offer
negotiate({undo_offer, Item}, S=#state{otheritems=OtherItems}) ->
    notice(S, "Other player cancelling offer on ~p", [Item]),
    {next_state, negotiate, S#state{otheritems=remove(Item, OtherItems)}};
```

This is an ugly aspect of using asynchronous messages on both sides. One set of message has the form 'make' and 'retract', while the other has 'do' and 'undo'. This is entirely arbitrary and only used to differentiate between player-to-FSM communications and FSM-to-FSM communications. Note that on those coming from our own player, we have to tell the other side about the changes we're making.

Another responsibility is to handle the `are_you_ready` message we mentioned in the protocol. This one is the last asynchronous event to handle in the `negotiate` state:

```
negotiate(are_you_ready, S=#state{other=OtherPid}) ->
    io:format("Other user ready to trade.~n"),
    notice(S,
        "Other user ready to transfer goods:~n"
        "You get ~p, The other side gets ~p",
        [S#state.otheritems, S#state.ownitems]),
    not_yet(OtherPid),
    {next_state, negotiate, S};
negotiate(Event, Data) ->
    unexpected(Event, negotiate),
    {next_state, negotiate, Data}.
```

As described in the protocol, whenever we're not in the `wait` state and receive this message, we must reply with `not_yet`. We're also outputting trade details to the user so a decision can be made.

When such a decision is made and the user is ready, the `ready` event will be sent. This one should be synchronous because we don't want the user to keep modifying his offer by adding items while claiming he's ready:

```
negotiate(ready, From, S = #state{other=OtherPid}) ->
    are_you_ready(OtherPid),
    notice(S, "asking if ready, waiting", []),
    {next_state, wait, S#state{from=From}};
negotiate(Event, _From, S) ->
```

```
unexpected(Event, negotiate),  
{next_state, negotiate, S}.
```

At this point a transition to the wait state should be made. Note that just waiting for the other is not interesting. We save the *From* variable so we can use it with `gen_fsm:reply/2` when we have something to tell to the client.

The wait state is a funny beast. New items might be offered and retracted because the other user might not be ready. It makes sense, then, to automatically rollback to the negotiating state. It would suck to have great items offered to us, only for the other to remove them and declare himself ready, stealing our loot. Going back to negotiation is a good decision:

```
wait({do_offer, Item}, S=#state{otheritems=OtherItems}) ->  
    gen_fsm:reply(S#state.from, offer_changed),  
    notice(S, "other side offering ~p", [Item]),  
    {next_state, negotiate, S#state{otheritems=add(Item, OtherItems)}};  
wait({undo_offer, Item}, S=#state{otheritems=OtherItems}) ->  
    gen_fsm:reply(S#state.from, offer_changed),  
    notice(S, "Other side cancelling offer of ~p", [Item]),  
    {next_state, negotiate, S#state{otheritems=remove(Item, OtherItems)}};
```

Now that's something meaningful and we reply to the player with the coordinates we stored in `S#state.from`. The next set of messages we need to worry about are those related to synchronising both FSMs so they can move to the ready state and confirm the trade. For this one we should really focus on the protocol defined earlier.



The three messages we could have are `are_you_ready` (because the other user just declared himself ready), `not_yet` (because we asked the other if he was ready and he was not) and `ready!` (because we asked the other if he was ready and he was).

We'll start with `are_you_ready`. Remember that in the protocol we said that there could be a race condition hidden there. The only thing we can do is send the `ready!` message with `am_ready/1` and deal with the rest later:

```
wait(are_you_ready, S=#state{}) ->
    am_ready(S#state.other),
    notice(S, "asked if ready, and I am. Waiting for same reply", []),
    {next_state, wait, S};
```

We'll be stuck waiting again, so it's not worth replying to our client yet. Similarly, we won't reply to the client when the other side sends a `not_yet` to our invitation:

```
wait(not_yet, S = #state{}) ->
    notice(S, "Other not ready yet", []),
    {next_state, wait, S};
```

On the other hand, if the other is ready, we send an extra `ready!` message to the other FSM, reply to our own user and then move to the `ready` state:

```

wait('ready!', S=#state{}) ->
    am_ready(S#state.other),
    ack_trans(S#state.other),
    gen_fsm:reply(S#state.from, ok),
    notice(S, "other side is ready. Moving to ready state", []),
    {next_state, ready, S};

%% DOn't care about these!
wait(Event, Data) ->
    unexpected(Event, wait),
    {next_state, wait, Data}.

```

You might have noticed that I've used `ack_trans/1`. In fact, both FSMs should use it. Why is this? To understand this we have to start looking at what goes on in the `ready!` state.



When in the `ready` state, both players' actions become useless (except cancelling). We won't care about new item offers. This gives us some liberty. Basically, both FSMs can freely talk to each other without worrying about the rest of the world. This lets us implement our bastardization of a two-phase commit. To begin this commit without either player acting, we'll need an event to trigger an action from the FSMs. The `ack` event from `ack_trans/1` is used for that. As soon as we're in the `ready` state, the message is treated and acted upon; the transaction can begin.

Two-phase commits require synchronous communications, though. This means we can't have both FSMs starting the transaction at once,

because they'll end up deadlocked. The secret is to find a way to decide that one finite state machine should initiate the commit, while the other will sit and wait for orders from the first one.

It turns out that the engineers and computer scientists who designed Erlang were pretty smart (well, we knew that already). The pids of any process can be compared to each other and sorted. This can be done no matter when the process was spawned, whether it's still alive or not, or if it comes from another VM (we'll see more about this when we get into distributed Erlang).

Knowing that two pids can be compared and one will be greater than the other, we can write a function priority/2 that will take two pids and tell a process whether it's been elected or not:

```
priority(OwnPid, OtherPid) when OwnPid > OtherPid -> true;
priority(OwnPid, OtherPid) when OwnPid < OtherPid -> false.
```

And by calling that function, we can have one process starting the commit and the other following the orders.

Here's what this gives us when included in the ready state, after receiving the ack message:

```
ready(ack, S=#state{}) ->
    case priority(self(), S#state.other) of
        true ->
            try
                notice(S, "asking for commit", []),
                ready_commit = ask_commit(S#state.other),
                notice(S, "ordering commit", []),
                ok = do_commit(S#state.other),
                notice(S, "committing...", []),
                commit(s),
                {stop, normal, S}
            catch Class:Reason ->
                %% abort! Either ready_commit or do_commit failed
```

```

    notice(S, "commit failed", []),
    {stop, {Class, Reason}, S}
end;
false ->
    {next_state, ready, S}
end;
ready(Event, Data) ->
    unexpected(Event, ready),
    {next_state, ready, Data}.

```

This big try ... catch expression is the leading FSM deciding how the commit works. Both `ask_commit/1` and `do_commit/1` are synchronous. This lets the leading FSM call them freely. You can see that the other FSM just goes and wait. It will then receive the orders from the leading process. The first message should be `ask_commit`. This is just to make sure both FSMs are still there; nothing wrong happened, they're both dedicated to completing the task:

```

ready(ask_commit, _From, S) ->
    notice(S, "replying to ask_commit", []),
    {reply, ready_commit, ready, S};

```

Once this is received, the leading process will ask to confirm the transaction with `do_commit`. That's when we must commit our data:

```

ready(do_commit, _From, S) ->
    notice(S, "committing...", []),
    commit(S),
    {stop, normal, ok, S};
ready(Event, _From, Data) ->
    unexpected(Event, ready),
    {next_state, ready, Data}.

```

And once it's done, we leave. The leading FSM will receive `ok` as a reply and will know to commit on its own end afterwards. This explains why we need the big try ... catch: if the replying FSM dies or its player cancels

the transaction, the synchronous calls will crash after a timeout. The commit should be aborted in this case.

Just so you know, I defined the commit function as follows:

```
commit(S = #state{}) ->
    io:format("Transaction completed for ~s.~n",
              "Items sent are:~n~p,~n received are:~n~p.~n",
              "This operation should have some atomic save~n",
              "in a database.~n",
              [S#state.name, S#state.ownitems, S#state.otheritems]).
```

Pretty underwhelming, eh? It's generally not possible to do a true safe commit with only two participants—a third party is usually required to judge if both players did everything right. If you were to write a true commit function, it should contact that third party on behalf of both players, and then do the safe write to a database for them or rollback the whole exchange. We won't go into such details and the current commit/1 function will be enough for the needs of this book.

We're not done yet. We have not yet covered two types of events: a player cancelling the trade and the other player's finite state machine crashing. The former can be dealt with by using the callbacks `handle_event/3` and `handle_sync_event/4`. Whenever the other user cancels, we'll receive an asynchronous notification:

```
%% The other player has sent this cancel event
%% stop whatever we're doing and shut down!
handle_event(cancel, _StateName, S=#state{}) ->
    notice(S, "received cancel event", []),
    {stop, other_cancelled, S};
handle_event(Event, StateName, Data) ->
    unexpected(Event, StateName),
    {next_state, StateName, Data}.
```

When we do it we must not forget to tell the other before quitting ourselves:

```

%% This cancel event comes from the client. We must warn the other
%% player that we have a quitter!
handle_sync_event(cancel, _From, _StateName, S = #state{}) ->
    notify_cancel(S#state.other),
    notice(S, "cancelling trade, sending cancel event", []),
    {stop, cancelled, ok, S};

%% Note: DO NOT reply to unexpected calls. Let the call-maker crash!
handle_sync_event(Event, _From, StateName, Data) ->
    unexpected(Event, StateName),
    {next_state, StateName, Data}.

```

And voilà! The last event to take care of is when the other FSM goes down. Fortunately, we had set a monitor back in the idle state. We can match on this and react accordingly:

```

handle_info({'DOWN', Ref, process, Pid, Reason}, _, S=#state{other=Pid, monitor=Ref}) ->
    notice(S, "Other side dead", []),
    {stop, {other_down, Reason}, S};
handle_info(Info, StateName, Data) ->
    unexpected(Info, StateName),
    {next_state, StateName, Data}.

```

Note that even if the cancel or DOWN events happen while we're in the commit, everything should be safe and nobody should get its items stolen.

Note: we used `io:format/2` for most of our messages to let the FSMs communicate with their own clients. In a real world application, we might want something more flexible than that. One way to do it is to let the client send in a Pid, which will receive the notices sent to it. That process could be linked to a GUI or any other system to make the player aware of the events. The `io:format/2` solution was chosen for its simplicity: we want to focus on the FSM and the asynchronous protocols, not the rest.

Only two callbacks left to cover! They're `code_change/4` and `terminate/3`. For now, we don't have anything to do with `code_change/4` and only

export it so the next version of the FSM can call it when it'll be reloaded. Our terminate function is also really short because we didn't handle real resources in this example:

```
code_change(_OldVsn, StateName, Data, _Extra) ->
{ok, StateName, Data}.
```

```
%% Transaction completed.
terminate(normal, ready, S=#state{}) ->
    notice(S, "FSM leaving.", []);
terminate(_Reason, _StateName, _StateData) ->
    ok.
```

Whew.

We can now try it. Well, trying it is a bit annoying because we need two processes to communicate to each other. To solve this, I've written the tests in the file `trade_calls.erl`, which can run 3 different scenarios. The first one is `main_ab/0`. It will run a standard trade and output everything. The second one is `main_cd/0` and will cancel the transaction halfway through. The last one is `main_ef/0` and is very similar to `main_ab/0`, except it contains a different race condition. The first and third tests should succeed, while the second one should fail (with a crapload of error messages, but that's how it goes). You can try it if you feel like it.

That Was Quite Something



If you've found this chapter a bit harder than the others, I must remind you that it's entirely normal. I've just gone crazy and decided to make something hard out of the generic finite-state machine behaviour. If you feel confused, ask yourself these questions: Can you understand how different events are handled depending on the state your process is in? Do you understand how you can transition from one state to the other? Do you know when to use `send_event/2` and `sync_send_event/2-3` as opposed to `send_all_state_event/2` and `sync_send_all_state_event/3`? If you answered yes to these questions, you understand what `gen_fsm` is about.

The rest of it with the asynchronous protocols, delaying replies and carrying the *From* variable, giving a priority to processes for synchronous calls, bastardized two-phase commits and whatnot *are not essential to understand*. They're mostly there to show what can be done and to highlight the difficulty of writing truly concurrent software, even in a language like Erlang. Erlang doesn't excuse you from planning or thinking, and Erlang won't solve your problems for you. It'll only give you tools.

That being said, if you understood everything about these points, you can be proud of yourself (especially if you had never written concurrent software before). You are now starting to really think concurrently.

Fit for the Real World?

In a real game, there is a lot more stuff going on that could make trading even more complex. Items could be worn by the characters and damaged by enemies while they're being traded. Maybe items could be moved in and out of the inventory while being exchanged. Are the players on the same server? If not, how do you synchronise commits to different databases?

Our trade system is sane when detached from the reality of any game. Before trying to fit it in a game (if you dare), make sure everything goes right. Test it, test it, and test it again. You'll likely find that testing concurrent and parallel code is a complete pain. You'll lose hair, friends and a piece of your sanity. Even after this, you'll have to know your system is always as strong as its weakest link and thus potentially very fragile nonetheless.

Don't Drink Too Much Kool-Aid:

While the model for this trade system seems sound, subtle concurrency bugs and race conditions can often rear their ugly heads a long time after they were written, and even if they've been running for years. While my code is generally bullet proof (yeah, right), you sometimes have to face swords and knives. Beware the dormant bugs.

Fortunately, we can put all of this madness behind us. We'll next see how OTP allows you to handle various events, such as alarms and logs, with the help of the gen_event behaviour.

Event Handlers

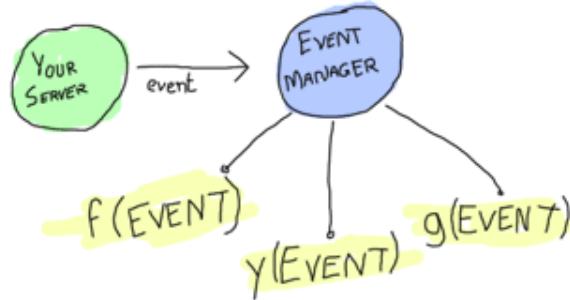
Handle This! *pumps shotgun*

There is a certain thing that I've avoided getting into in a few of the previous examples. If you look back at the [reminder app](#), you'll see that I somehow mentioned that we could notify clients, whether they were IM, mail, etc. In the [previous chapter](#), our trading system used `io:format/2` to notify people of what was going on.

You can probably see the common link between both cases. They're all about letting people (or some process or application) know about an event that happened at some point in time. In one case, we only output the results while in the other, we took the Pid of subscribers before sending them a message.

The output approach is minimalist and can not be extended with ease. The one with subscribers is certainly valid. In fact, it's pretty useful when each of the subscribers has a long-running operation to do after receiving an event. In simpler cases, where you do not necessarily want a standby process waiting for events for each of the callbacks, a third approach can be taken.

This third approach simply takes a process which accepts functions and lets them run on any incoming event. This process is usually called an *event manager* and it might end up looking a bit like this:



Doing things that way has a few advantages:

- If your server has many subscribers, it can keep going because it only needs to forward events once
- If there is a lot of data to be transferred, it's only done once and all callbacks operate on that same instance of the data
- You don't need to spawn processes for short lived tasks

And of course there are a few downsides too:

- If all functions need to run for a long time, they're going to block each other. This can be prevented by actually having the function forward the event to a process, basically turning the event manager as an event forwarder (something similar to what we did for the reminder app)
- In fact, a function that loops indefinitely can prevent any new event from being handled until something crashes.

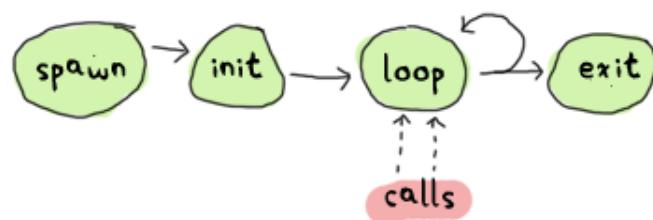
There is a way to solve these downsides, which is a bit underwhelming. Basically, you have to turn the event manager approach into the subscriber one. Luckily, the event manager approach is flexible enough to do it with ease and we'll see how to do it later in this chapter.

I usually write a very basic version of the OTP behaviour we'll see in pure Erlang beforehand, but in this case, I'll instead come straight to the point. Here comes gen_event.

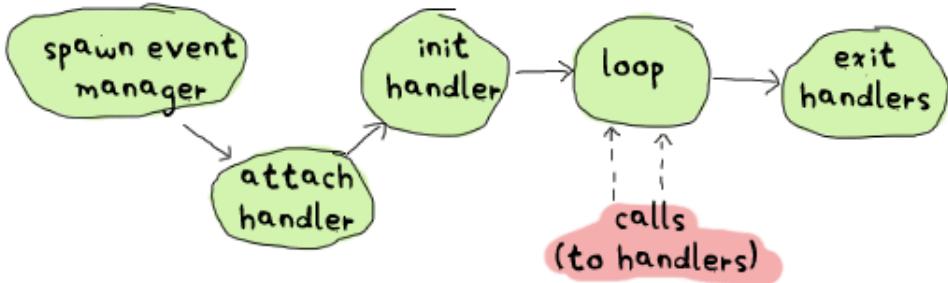
Generic Event Handlers

The gen_event behaviour differs quite a bit from the gen_server and gen_fsm behaviours in that you are never really starting a process. The whole part I've described above about 'accepting a callback' is the reason for this. The gen_event behaviour basically runs the process that accepts and calls functions, and you only provide a module with these functions. This is to say, you have nothing to do with regards to event manipulation except give your callback functions in a format that pleases the *event manager*. All managing is done for free; you only provide what's specific to your application. This is not really surprising given OTP is, again, all about separating what's generic from specific.

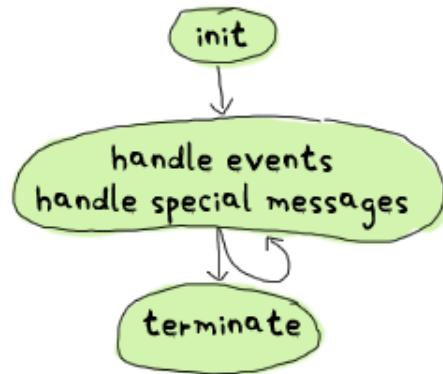
This separation, however, means that the standard spawn -> init -> loop -> terminate pattern will only be applied to event handlers. Now if you recall what has been said before, event handlers are a bunch of functions running in the manager. This means the currently presented model:



Switches to something more like this for the programmer:



Each event handler can hold its own state, carried around by the manager for them. Each event handler can then take the form:



This is nothing too complex so let's get on with the event handlers' callbacks themselves.

init and terminate

The init and terminate functions are similar to what we've seen in the previous behaviours with servers and finit state machines. init/1 takes a list of arguments and returns {ok, State}. Whatever happens in init/1 should have its counterpart in terminate/2.

handle_event

The handle_event(Event, State) function is more or less the core of gen_event's callback modules. It works like gen_server's handle_cast/2

in that it works asynchronously. It differs with regards to what it can return though:

- {ok, NewState}
- {ok, NewState, hibernate}, which puts the event manager itself into hibernation until the next event
- remove_handler
- {swap_handler, Args1, NewState, NewHandler, Args2}



The tuple {ok, NewState} works in a way similar to what we've seen with `gen_server:handle_cast/2`; it simply updates its own state and doesn't reply to anyone. In the case of {ok, NewState, hibernate} it is to be noted that the whole event manager is going to be put in hibernation. Remember that event handlers run in the same process as their manager. Then `remove_handler` drops the handler from the manager. This can be useful whenever your event handler knows its done and it has nothing else to do. Finally, there's {swap_handler, Args1, NewState, NewHandler, Args2}. This one is not used too frequently, but what it does is remove the current event handler and replace it with a new one. This is done by first calling `CurrentHandler:terminate(Args1, NewState)` and removing the current handler, then adding a new one by calling `NewHandler:init(Args2,`

`ResultFromTerminate`). This can be useful in the cases where you know some specific event happened and you're better off giving control to a new handler. This is likely the kind of thing where you'll simply know when you need it. Again, it's not that frequently used.

All incoming events can come from `gen_event:notify/2` which is asynchronous like `gen_server:cast/2` is. There is also `gen_event:sync_notify/2` which is synchronous. This is a bit funny to say, because `handle_event/2` remains asynchronous. The idea here is that the function call only returns once all event handlers have seen and treated the new message. Until then, the event manager will keep blocking the calling process by not replying.

handle_call

This is similar to a `gen_server`'s `handle_call` callback, except that it can return `{ok, Reply, NewState}`, `{ok, Reply, NewState, hibernate}`, `{remove_handler, Reply}` or `{swap_handler, Reply, Args1, NewState, Handler2, Args2}`. The `gen_event:call/3-4` function is used to make the call.

This raises a question. How does this work when we have something like 15 different event handlers? Do we expect 15 replies or just one that contains them all? Well, in fact we'll be forced to choose only one handler to reply to us. We'll get into the details of how this is done when we actually see how to attach handlers to our event manager, but if you're impatient, you can look at how the function `gen_event:add_handler/3` works to try to figure it out.

handle_info

The `handle_info/2` callback is pretty much the same as `handle_event` (same return values and everything), with the exception that it only treats out of band messages, such as exit signals, messages sent directly to the event manager with the `!` operator, etc. It has use cases similar to those of `handle_info` in `gen_server` and in `gen_fsm`.

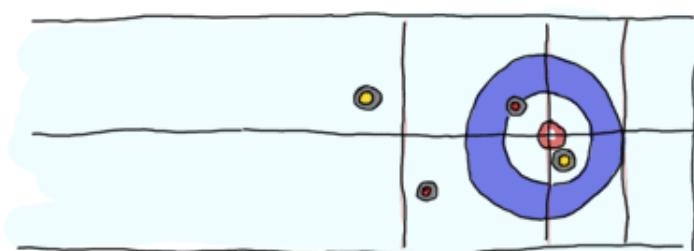
code_change

Code change works in exactly the same manner as it does for `gen_servers`, except it's for each individual event handler. It takes 3 arguments, `OldVsn`, `State`, and `Extra`, which are in order, the version number, the current handler's state and data we can ignore for now. All it needs to do is return `{ok, NewState}`.

It's Curling Time!

With the callbacks seen, we can start looking at implementing something with `gen_event`. For this part of the chapter, I've chosen to make a set of event handlers used to track game updates of one of the most entertaining sports in the world: curling.

If you've never seen or played curling before (which is a shame!), the rules are relatively simple:



You have two teams and they try to send a curling stone sliding on the ice, trying to get to the middle of the red circle. They do this with 16 stones and the team with the stone closest to the center wins a point at the end of the round (called an *end*). If the team has the two closest stones, it earns two points, and so on. There are 10 ends and the team with the most points at the end of the 10 ends wins the game.

There are more rules making the game more fascinating, but this is a book on Erlang, not extremely fascinating winter sports. If you want to learn more about the rules, I suggest you head up to the Wikipedia article on curling.

For this entirely real-world-relevant scenario, we'll be working for the next winter Olympic Games. The city where everything happens are just done building the arena where the matches will take place and they're working on getting the scoreboard ready. It turns out that we have to program a system that will let some official enter game events, such as when a stone has been thrown, when a round ends or when a game is over, and then route these events to the scoreboard, a stats system, news reporters' feeds, etc.

Being as clever as we are, we know this is a chapter on `gen_event` and deduce we will likely use it to accomplish our task. We won't implement all the rules given this is more of an example, but feel free to do so when we're done with the chapter. I promise not to be mad.

We'll start with the scoreboard. Because they're installing it right now, we'll make use of a fake module that would usually let us interact with it, but for now it'll only use standard output to show

what's going on. This is where `curling_scoreboard_hw.erl` comes in:

```
-module(curling_scoreboard_hw).
-export([add_point/1, next_round/0, set_teams/2, reset_board/0]).  
  
%% This is a 'dumb' module that's only there to replace what a real hardware  
%% controller would likely do. The real hardware controller would likely hold  
%% some state and make sure everything works right, but this one doesn't mind.  
  
%% Shows the teams on the scoreboard.  
set_teams(TeamA, TeamB) ->  
    io:format("Scoreboard: Team ~s vs. Team ~s~n", [TeamA, TeamB]).  
  
next_round() ->  
    io:format("Scoreboard: round over~n").  
  
add_point(Team) ->  
    io:format("Scoreboard: increased score of team ~s by 1~n", [Team]).  
  
reset_board() ->  
    io:format("Scoreboard: All teams are undefined and all scores are 0~n").
```

So this is all the functionality the scoreboard has. They usually have a timer and other awesome functionalities, but whatever. Seems like the Olympics committee didn't feel like having us implementing trivialities for a tutorial.

This hardware interface lets us have a little bit of design time to ourselves. We know that there are a few events to handle for now: adding teams, going to the next round, setting the number of points. We will only use the `reset_board` functionality when starting a new game and won't need it as part of our protocol. The events we need might take the following form in our protocol:

- {set_teams, TeamA, TeamB}, where this is translated to a single call to curling_scoreboard_hw:set_teams(TeamA, TeamB);
- {add_points, Team, N}, where this is translated to N calls to curling_scoreboard_hw:add_point(Team);
- next_round, which gets translated to a single call with the same name.

We can start our implementation with this basic event handler skeleton:

```
-module(curling_scoreboard).
-behaviour(gen_event).

-export([init/1, handle_event/2, handle_call/2, handle_info/2, code_change/3,
        terminate/2]).

init([]) ->
    {ok, []}.

handle_event(_, State) ->
    {ok, State}.

handle_call(_, State) ->
    {ok, ok, State}.

handle_info(_, State) ->
    {ok, State}.

code_change(_OldVsn, State, _Extra) ->
    {ok, State}.

terminate(_Reason, _State) ->
    ok.
```

This is a skeleton that we can use for every gen_event callback module out there. For now, the scoreboard event handler itself won't need to do anything special except forward the calls to the hardware module. We expect the events to come from gen_event:notify/2, so the handling of the protocol should be done in handle_event/2. The file curling_scoreboard.erl shows the updates:

```
-module(curling_scoreboard).
-behaviour(gen_event).

-export([init/1, handle_event/2, handle_call/2, handle_info/2, code_change/3,
        terminate/2]).

init([]) ->
    {ok, []}.

handle_event({set_teams, TeamA, TeamB}, State) ->
    curling_scoreboard_hw:set_teams(TeamA, TeamB),
    {ok, State};

handle_event({add_points, Team, N}, State) ->
    [curling_scoreboard_hw:add_point(Team) || _ <- lists:seq(1,N)],
    {ok, State};

handle_event(next_round, State) ->
    curling_scoreboard_hw:next_round(),
    {ok, State};

handle_event(_, State) ->
    {ok, State}.

handle_call(_, State) ->
    {ok, ok, State}.

handle_info(_, State) ->
    {ok, State}.
```

You can see the updates done to the handle_event/2 function.

Trying it:

```
1> c(curling_scoreboard_hw).
{ok,curling_scoreboard_hw}
2> c(curling_scoreboard).
{ok,curling_scoreboard}
3> {ok, Pid} = gen_event:start_link().
{ok,<0.43.0>}
4> gen_event:add_handler(Pid, curling_scoreboard, []).
ok
5> gen_event:notify(Pid, {set_teams, "Pirates", "Scotsmen"}).
Scoreboard: Team Pirates vs. Team Scotsmen
ok
6> gen_event:notify(Pid, {add_points, "Pirates", 3}).
ok
Scoreboard: increased score of team Pirates by 1
Scoreboard: increased score of team Pirates by 1
Scoreboard: increased score of team Pirates by 1
7> gen_event:notify(Pid, next_round).
Scoreboard: round over
ok
8> gen_event:delete_handler(Pid, curling_scoreboard, turn_off).
ok
9> gen_event:notify(Pid, next_round).
ok
```

A few things are going on here. The first of them is that we're starting the gen_event process as a standalone thing. We then attach our event handler to it dynamically with gen_event:add_handler/3. This can be done as many times as you want. However, as mentioned in the handle_call part earlier, this might cause problems when you want to work with a particular event handler. If you want to call, add or delete a specific handler when there's more than one instance of it, you'll have to find a

way to uniquely identify it. My favorite way of doing it (one that works great if you don't have anything more specific in mind) is to just use `make_ref()` as a unique value. To give this value to the handler, you add it by calling `add_handler/3` as `gen_event:add_handler(Pid, {Module, Ref}, Args)`. From this point on, you can use `{Module, Ref}` to talk to that specific handler. Problem solved.



Anyway, you can then see that we send messages to the event handler, which successfully calls the hardware module. We then remove the handler. Here, `turn_off` is an argument to the `terminate/2` function, which our implementation currently doesn't care about. The handler is gone, but we can still send events to the event manager. Hooray.

One awkward thing with the code snippet above is that we're forced to call the `gen_event` module directly and show everyone what our protocol looks like. A better option would be to provide an abstraction module on top of it that just wraps all we need. This will look a lot nicer to everyone using our code and will, again, let us change the implementation if (or when) we need to do it. It will also let us specify what handlers are necessary to include for a standard curling game:

```
-module(curling).  
-export([start_link/2, set_teams/3, add_points/3, next_round/1]).
```

```
start_link(TeamA, TeamB) ->
    {ok, Pid} = gen_event:start_link(),
    %% The scoreboard will always be there
    gen_event:add_handler(Pid, curling_scoreboard, []),
    set_teams(Pid, TeamA, TeamB),
    {ok, Pid}.
```

```
set_teams(Pid, TeamA, TeamB) ->
    gen_event:notify(Pid, {set_teams, TeamA, TeamB}).
```

```
add_points(Pid, Team, N) ->
    gen_event:notify(Pid, {add_points, Team, N}).
```

```
next_round(Pid) ->
    gen_event:notify(Pid, next_round).
```

And now running it:

```
1> c(curling).
{ok,curling}
2> {ok, Pid} = curling:start_link("Pirates", "Scotsmen").
Scoreboard: Team Pirates vs. Team Scotsmen
{ok,<0.78.0>}
3> curling:add_points(Pid, "Scotsmen", 2).
Scoreboard: increased score of team Scotsmen by 1
Scoreboard: increased score of team Scotsmen by 1
ok
4> curling:next_round(Pid).
Scoreboard: round over
ok
```



This doesn't look like much of an advantage, but it's really about making it nicer to use that code (and reduces the possibilities of writing the messages wrong).

Alert the Press!

We've got the basic scoreboard done, now we want international reporters to be able to get live data from our official in charge of updating our system. Because this is an example program, we won't go through the steps of setting up a socket and writing a protocol for the updates, but we'll put the system in place to do it by putting an intermediary process in charge of it.

Basically, whenever a news organization feels like getting into the game feed, they'll register their own handler that just forwards them the data they need. We'll effectively going to turn our gen_event server into some kind of message hub, just routing them to whoever needs them.

The first thing to do is update the curling.erl module with the new interface. Because we want things to be easy to use, we'll only add two functions, `join_feed/2` and `leave_feed/2`. Joining the feed

should be doable just by inputting the right Pid for the event manager and the Pid to forward all the events to. This should return a unique value that can then be used to unsubscribe from the feed with `leave_feed/2`:

```
%% Subscribes the pid ToPid to the event feed.  
%% The specific event handler for the newsfeed is  
%% returned in case someone wants to leave  
join_feed(Pid, ToPid) ->  
    HandlerId = {curling_feed, make_ref()},  
    gen_event:add_handler(Pid, HandlerId, [ToPid]),  
    HandlerId.  
  
leave_feed(Pid, HandlerId) ->  
    gen_event:delete_handler(Pid, HandlerId, leave_feed).
```

Note that I'm using the technique described earlier for multiple handlers (`{curling_feed, make_ref()}`). You can see that this function expects a `gen_event` callback module named `curling_feed`. If I only used the name of the module as a `HandlerId`, things would have still worked fine, except that we would have no control on which handler to delete when we're done with one instance of it. The event manager would just pick one of them in an undefined manner. Using a Ref makes sure that some guy from the Head-Smashed-In Buffalo Jump press leaving the place won't disconnect a journalist from *The Economist* (no idea why they'd do a report on curling, but what do you know). Anyway, here is the implementation I've made of the `curling_feed` module:

```
-module(curling_feed).  
-behaviour(gen_event).  
  
-export([init/1, handle_event/2, handle_call/2, handle_info/2, code_change/3,  
        terminate/2]).
```

```

init([Pid]) ->
    {ok, Pid}.

handle_event(Event, Pid) ->
    Pid ! {curling_feed, Event},
    {ok, Pid}.

handle_call(_, State) ->
    {ok, ok, State}.

handle_info(_, State) ->
    {ok, State}.

code_change(_OldVsn, State, _Extra) ->
    {ok, State}.

terminate(_Reason, _State) ->
    ok.

```

The only interesting thing here is still the `handle_event/2` function, which blindly forwards all events to the subscribing Pid. Now when we use the new modules:

```

1> c(curling), c(curling_feed).
{ok,curling_feed}
2> {ok, Pid} = curling:start_link("Saskatchewan Roughriders", "Ottawa Roughriders").
Scoreboard: Team Saskatchewan Roughriders vs. Team Ottawa Roughriders
{ok,<0.165.0>}
3> HandlerId = curling:join_feed(Pid, self()).
{curling_feed,#Ref<0.0.0.909>}
4> curling:add_points(Pid, "Saskatchewan Roughriders", 2).
Scoreboard: increased score of team Saskatchewan Roughriders by 1
ok
Scoreboard: increased score of team Saskatchewan Roughriders by 1
5> flush().

```

```
Shell got {curling_feed,{add_points,"Saskatchewan Roughriders",2}}
```

ok

```
6> curling:leave_feed(Pid, HandlerId).
```

ok

```
7> curling:next_round(Pid).
```

Scoreboard: round over

```
ok
```

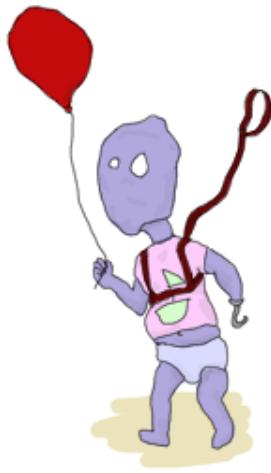
```
8> flush().
```

ok

And we can see that we added ourselves to the feed, got the updates, then left and stopped receiving them. You can actually try to add many processes many times and it will work fine.

This introduces a problem though. What if one of the curling feed subscribers crashes? Do we just keep the handler going on there? Ideally, we wouldn't have to. In fact, we don't have to. All that needs to be done is to change the call from gen_event:add_handler/3 to gen_event:add_sup_handler/3. If you crash, the handler is gone. Then on the opposite end, if the gen_event manager crashes, the message {gen_event_EXIT, Handler, Reason} is sent back to you so you can handle it. Easy enough, right? Think again.

Don't Drink Too Much Kool-Aid



It might have happened at some time in your childhood that you went to your aunt or grandmother's place for a party or something. If you were mischievous in any way, you would have several adults looking over you, on top of your parents. Now if you ever did something wrong, you would get scolded by your mom, dad, aunt, grandmother and then everyone would keep telling you after that even though you already clearly knew you had done something wrong. Well `gen_event:add_sup_handler/3` is a bit like that; no, seriously.

Whenever you use `gen_event:add_sup_handler/3`, a link is set up between your process and the event manager so both of them are supervised and the handler knows if its parent process fails. If you recall the [Errors and Processes](#) chapter and its section on monitors, I have mentioned that monitors are *great for writing libraries which need to know what's going on with other processes* because they can be stacked, at the opposite of links. Well `gen_event` predates the appearance of monitors in Erlang and a strong commitment to backwards compatibility introduced this pretty bad wart. Basically, because you could have the same process acting as the parent of many event handlers, so the

library doesn't ever unlink the processes (except when it terminates for good) just in case. Monitors would actually solve the problem, but they aren't being used there.

This means that everything goes alright when your own process crashes: the supervised handler is terminated (with the call to `YourModule:terminate({stop, Reason}, State)`). Everything goes alright when your handler itself crashes (but not the event manager): you will receive `{gen_event_EXIT, HandlerId, Reason}`. When the event manager is shut down though, you will either:

- Receive the `{gen_event_EXIT, HandlerId, Reason}` message then crash because you're not trapping exits;
- Receive the `{gen_event_EXIT, HandlerId, Reason}` message, then a standard 'EXIT' message that is either superfluous or confusing.

That's quite a wart, but at least you know about it. You can try and switch your event handler to a supervised one if you feel like it. It'll be safer even if it risks being more annoying in some cases. Safety first.

We're not done yet! what happens if some member of the media is not there on time? We need to be able to tell them from the feed what the current state of the game is. For this, we'll write an additional event handler named `curling_accumulator`. Again, before writing it entirely, we might want to add it to the `curling` module with the few calls we want:

```
-module(curling).  
-export([start_link/2, set_teams/3, add_points/3, next_round/1]).  
-export([join_feed/2, leave_feed/2]).  
-export([game_info/1]).
```

```

start_link(TeamA, TeamB) ->
    {ok, Pid} = gen_event:start_link(),
    %% The scoreboard will always be there
    gen_event:add_handler(Pid, curling_scoreboard, []),
    %% Start the stats accumulator
    gen_event:add_handler(Pid, curling_accumulator, []),
    set_teams(Pid, TeamA, TeamB),
    {ok, Pid}.

```

%% skipping code here

```

%% Returns the current game state.
game_info(Pid) ->
    gen_event:call(Pid, curling_accumulator, game_data).

```

A thing to notice here is that the game_info/1 function uses only curling_accumulator as a handler id. In the cases where you have many versions of the same handler, the hint about using make_ref() (or any other means) to ensure you write to the right handler still holds. Also note that I made the curling_accumulator handler start automatically, much like the scoreboard. Now for the module itself. It should be able to hold state for the curling game: so far we have teams, score and rounds to track. This can all be held in a state record, changed on each event received. Then, we will only need to reply to the game_data call, as below:

```

-module(curling_accumulator).
-behaviour(gen_event).

-export([init/1, handle_event/2, handle_call/2, handle_info/2, code_change/3,
        terminate/2]).

-record(state, {teams=orddict:new(), round=0}).

```

```

init([]) ->
    {ok, #state{}}.

handle_event({set_teams, TeamA, TeamB}, S=#state{teams=T}) ->
    Teams = orddict:store(TeamA, 0, orddict:store(TeamB, 0, T)),
    {ok, S#state{teams=Teams}};
handle_event({add_points, Team, N}, S=#state{teams=T}) ->
    Teams = orddict:update_counter(Team, N, T),
    {ok, S#state{teams=Teams}};
handle_event(next_round, S=#state{}) ->
    {ok, S#state{round = S#state.round+1}};
handle_event(_Event, Pid) ->
    {ok, Pid}.

handle_call(game_data, S=#state{teams=T, round=R}) ->
    {ok, {orddict:to_list(T), {round, R}}, S};
handle_call(_, State) ->
    {ok, ok, State}.

handle_info(_, State) ->
    {ok, State}.

code_change(_OldVsn, State, _Extra) ->
    {ok, State}.

terminate(_Reason, _State) ->
    ok.

```

So you can see we basically just update the state until someone asks for game details, at which point we'll be sending them back. We did this in a very basic way. A perhaps smarter way to organize the code would have been to simply keep a list of all the events to ever happen in the game so we could send them back at once each time a new process subscribes to the feed. This

won't be needed here to show how things work, so let's focus on using our new code:

```
1> c(curling), c(curling_accumulator).
{ok,curling_accumulator}
2> {ok, Pid} = curling:start_link("Pigeons", "Eagles").
Scoreboard: Team Pigeons vs. Team Eagles
{ok,<0.242.0>}
3> curling:add_points(Pid, "Pigeons", 2).
Scoreboard: increased score of team Pigeons by 1
ok
Scoreboard: increased score of team Pigeons by 1
4> curling:next_round(Pid).
Scoreboard: round over
ok
5> curling:add_points(Pid, "Eagles", 3).
Scoreboard: increased score of team Eagles by 1
ok
Scoreboard: increased score of team Eagles by 1
Scoreboard: increased score of team Eagles by 1
6> curling:next_round(Pid).
Scoreboard: round over
ok
7> curling:game_info(Pid).
{[{"Eagles",3}, {"Pigeons",2}], {round,2}}
```

Enthralling! Surely the Olympic committee will love our code. We can pat ourselves on the back, cash in a fat check and go play videogames all night now.

We haven't seen all there is to do with gen_event as a module. In fact, we haven't seen the most common use of event handlers: logging and system alarms. I decided against showing them because pretty much any other source on Erlang out there uses

`gen_event` strictly for that. If you're interested in going there, check out `error_logger` first.

Even if we've not seen the most common uses of `gen_event`, it's important to say that we've seen all the concepts necessary to understanding them, building our own and integrating them into our applications. More importantly, we've finally covered the three main OTP behaviours used in active code development. We still have a few behaviours left to visit—those that act as a bunch of glue between all of our worker processes—such as the supervisor.

Who Supervises The Supervisors?

From Bad to Good



Supervisors are one of the most useful part of OTP you'll get to use. We've seen basic supervisors back in [Errors and Processes](#) and in [Designing a Concurrent Application](#). We've seen them as a way to keep our software going in case of errors by just restarting the faulty processes.

To be more detailed, our supervisors would start a *worker* process, link to it, and trap exit signals with `process_flag(trap_exit,true)` to know when the process died and restart it. This is fine when we want restarts, but it's also pretty dumb. Let's imagine that you're using the remote control to turn the TV on. If it doesn't work the first time, you might try once or twice just in case you didn't press right or the signal went wrong. Our supervisor, if it was trying to turn that very TV on, would keep trying forever, even if it turned out that the remote had no batteries or didn't even fit the TV. A pretty dumb supervisor.

Something else that was dumb about our supervisors is that they could only watch one worker at a time. Don't get me wrong, it's sometimes useful to have one supervisor for a single worker, but in large applications, this would mean you could only have a chain of supervisors, not a tree. How would you supervise a task where you need 2 or 3 workers at once? With our implementation, it just couldn't be done.

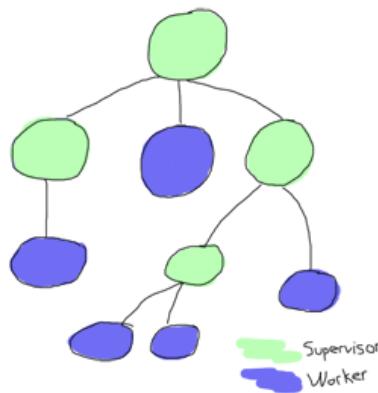
The OTP supervisors, fortunately, provide the flexibility to handle such cases (and more). They let you define how many times a worker should be restarted in a given period of time before giving up. They let you have more than one worker per supervisor and even let you pick between a few patterns to determine how they should depend on each other in case of a failure.

Supervisor Concepts

Supervisors are one of the simplest behaviours to use and understand, but one of the hardest behaviours to write a good design with. There are various strategies related to supervisors and application design, but before going there we need to understand more basic concepts because otherwise it's going to be pretty hard.

One of the words I have used in the text so far without much of a definition is the word 'worker'. Workers are defined a bit in opposition of supervisors. If supervisors are supposed to be processes which do nothing but make sure their children are restarted when they die, workers are processes in charge of doing actual work, and that may die while doing so. They are usually not trusted.

Supervisors can supervise workers and other supervisors, while workers should never be used in any position except under another supervisor:

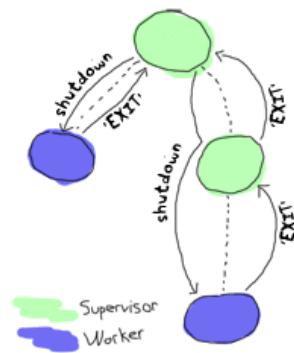


Why should every process be supervised? Well the idea is simple: if for some reason you're spawning unsupervised processes, how can you be sure they are gone or not? If you can't measure something, it doesn't exist. Now if a process exists in the void away from all your supervision trees, how do you know it exists or not? How did it get there? Will it happen again?

If it does happen, you'll find yourself leaking memory very slowly. So slowly your VM might suddenly die because it no longer has memory, and so slowly you

might not be able to easily track it until it happens again and again. Of course, you might say "If I take care and know what I'm doing, things will be fine". Maybe they will be fine, yeah. Maybe they won't. In a production system, you don't want to be taking chances, and in the case of Erlang, it's why you have garbage collection to begin with. Keeping things supervised is pretty useful.

Another reason why it's useful is that it allows to terminate applications in good order. It will happen that you'll write Erlang software that is not meant to run forever. You'll still want it to terminate cleanly though. How do you know everything is ready to be shut down? With supervisors, it's easy. Whenever you want to terminate an application, you have the top supervisor of the VM shut down (this is done for you with functions like `init:stop/1`). Then that supervisor asks each of its children to terminate. If some of the children are supervisors, they do the same:



This gives you a well-ordered VM shutdown, something that is very hard to do without having all of your processes being part of the tree.

Of course, there are times where your process will be stuck for some reason and won't terminate correctly. When that happens, supervisors have a way to brutally kill the process.

This is it for the basic theory of supervisors. We have workers, supervisors, supervision trees, different ways to specify dependencies, ways to tell supervisors when to give up on trying or waiting for their children, etc. This is not all that supervisors can do, but for now, this will let us cover the basic content required to actually use them.

Using Supervisors

This has been a very violent chapter so far: parents spend their time binding their children to trees, forcing them to work before brutally killing them. We wouldn't be real sadists without actually implementing it all though.

When I said supervisors were simple to use, I wasn't kidding. There is a single callback function to provide: `init/1`. It takes some arguments and that's about it. The catch is that it returns quite a complex thing. Here's an example return from a supervisor:

```
{ok, {{one_for_all, 5, 60},  
      [{fake_id,  
       {fake_mod, start_link, [SomeArg]},  
        permanent,  
        5000,  
        worker,  
        [fake_mod]},  
       {other_id,  
        {event_manager_mod, start_link, []},  
        transient,  
        infinity,  
        worker,  
        dynamic]}]}.
```

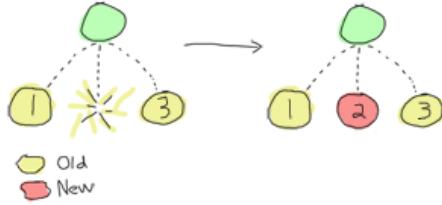
Say what? Yeah, that's pretty complex. A general definition might be a bit simpler to work with:

```
{ok, {{RestartStrategy, MaxRestart, MaxTime}, [ChildSpecs]}}.
```

Where *ChildSpec* stands for a child specification. *RestartStrategy* can be any of `one_for_one`, `rest_for_one`, `one_for_all` and `simple_one_for_one`.

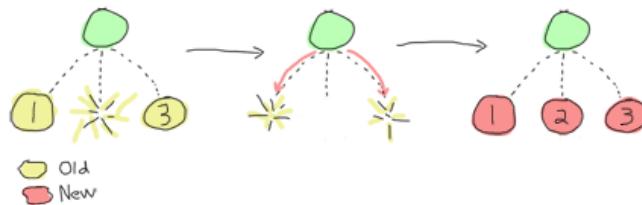
one_for_one

One for one is an intuitive restart strategy. It basically means that if your supervisor supervises many workers and one of them fails, only that one should be restarted. You should use `one_for_one` whenever the processes being supervised are independent and not really related to each other, or when the process can restart and lose its state without impacting its siblings.



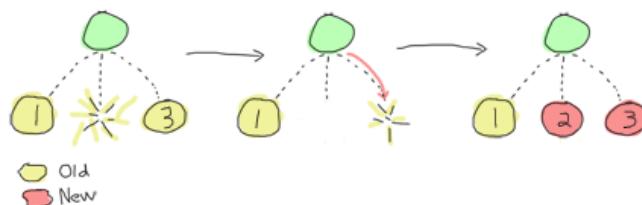
one_for_all

One for all has little to do with musketeers. It's to be used whenever all your processes under a single supervisor heavily depend on each other to be able to work normally. Let's say you have decided to add a supervisor on top of the trading system we implemented back in the [Rage Against The Finite State Machines](#) chapter. It wouldn't actually make sense to restart only one of the two traders if one of them crashed because their state would be out of sync. Restarting both of them at once would be a saner choice and `one_for_all` would be the strategy for that.



rest_for_one

This is a more specific kind of strategy. Whenever you have to start processes that depend on each other in a chain (*A* starts *B*, which starts *C*, which starts *D*, etc.), you can use `rest_for_one`. It's also useful in the case of services where you have similar dependencies (*X* works alone, but *Y* depends on *X* and *Z* depends on both). What a `rest_for_one` restarting strategy does, basically, is make it so if a process dies, all the ones that were started after it (depend on it) get restarted, but not the other way around.



simple_one_for_one

The `simple_one_for_one` restart strategy isn't the most simple one. We'll see it in more details when we get to use it, but it basically makes it so it takes only one kind of children, and it's to be used when you want to dynamically add them to the supervisor, rather than having them started statically.

To say it a bit differently, a `simple_one_for_one` supervisor just sits around there, and it knows it can produce one kind of child only. Whenever you want a new one, you ask for it and you get it. This kind of thing could theoretically be done with the standard `one_for_one` supervisor, but there are practical advantages to using the simple version.

Note: one of the big differences between `one_for_one` and `simple_one_for_one` is that `one_for_one` holds a list of all the children it has (and had, if you don't clear it), started in order, while `simple_one_for_one` holds a single definition for all its children and works using a dict to hold its data. Basically, when a process crashes, the `simple_one_for_one` supervisor will be much faster when you have a large number of children.

Restart limits

The last part of the `RestartStrategy` tuple is the couple of variables `MaxRestart` and `MaxTime`. The idea is basically that if more than `MaxRestarts` happen within `MaxTime` (in seconds), the supervisor just gives up on your code, shuts it down then kills itself to never return (that's how bad it is). Fortunately, that supervisor's supervisor might still have hope in its children and start them all over again.

Child Specifications

And now for the `ChildSpec` part of the return value. `ChildSpec` stands for `Child Specification`. Earlier we had the following two `ChildSpecs`:

```
[{fake_id,  
 {fake_mod, start_link, [SomeArg]},  
 permanent,  
 5000,  
 worker,  
 [fake_mod]},  
{other_id,  
 {event_manager_mod, start_link, []},  
 transient,
```

```
infinity,  
worker,  
dynamic}]
```

The child specification can be described in a more abstract form as:

```
{ChildId, StartFunc, Restart, Shutdown, Type, Modules}.
```

ChildId

The *ChildId* is just an internal name used by the supervisor internally. You will rarely need to use it yourself, although it might be useful for debugging purposes and sometimes when you decide to actually get a list of all the children of a supervisor. Any term can be used for the Id.

StartFunc

StartFunc is a tuple that tells how to start the child. It's the standard {M,F,A} format we've used a few times already. Note that it is very important that the starting function here is OTP-compliant and links to its caller when executed (hint: use `gen_*:start_link()` wrapped in your own module, all the time).

Restart

Restart tells the supervisor how to react when that particular child dies. This can take three values:

- permanent
- temporary
- transient

A permanent process should always be restarted, no matter what. The supervisors we implemented in our previous applications used this strategy only. This is usually used by vital, long-living processes (or services) running on your node.

On the other hand, a temporary process is a process that should never be restarted. They are for short-lived workers that are expected to fail and which have few bits of code who depend on them.

Transient processes are a bit of an in-between. They're meant to run until they terminate normally and then they won't be restarted. However, if they die of

abnormal causes (exit reason is anything but normal), they're going to be restarted. This restart option is often used for workers that need to succeed at their task, but won't be used after they do so.

You can have children of all three kinds mixed under a single supervisor. This might affect the restart strategy: a `one_for_all` restart won't be triggered by a temporary process dying, but that temporary process might be restarted under the same supervisor if a permanent process dies first!

Shutdown

Earlier in the text, I mentioned being able to shut down entire applications with the help of supervisors. This is how it's done. When the top-level supervisor is asked to terminate, it calls `exit(ChildPid, shutdown)` on each of the Pids. If the child is a worker and trapping exits, it'll call its own `terminate` function. Otherwise, it's just going to die. When a supervisor gets the `shutdown` signal, it will forward it to its own children the same way.

The `Shutdown` value of a child specification is thus used to give a deadline on the termination. On certain workers, you know you might have to do things like properly close files, notify a service that you're leaving, etc. In these cases, you might want to use a certain cutoff time, either in milliseconds or infinity if you are really patient. If the time passes and nothing happens, the process is then brutally killed with `exit(Pid, kill)`. If you don't care about the child and it can pretty much die without any consequences without any timeout needed, the atom `brutal_kill` is also an acceptable value. `brutal_kill` will make it so the child is killed with `exit(Pid, kill)`, which is untrappable and instantaneous.

Choosing a good `Shutdown` value is sometimes complex or tricky. If you have a chain of supervisors with `Shutdown` values like: 5000 → 2000 → 5000 → 5000, the two last ones will likely end up brutally killed, because the second one had a shorter cutoff time. It is entirely application dependent, and few general tips can be given on the subject.

Note: it is important to note that `simple_one_for_one` children are *not* respecting this rule with the `Shutdown` time. In the case of `simple_one_for_one`, the supervisor will just exit and it will be left to each of the workers to terminate on their own, after their supervisor is gone.

Type

Type simply lets the supervisor know whether the child is a worker or a supervisor. This will be important when upgrading applications with more advanced OTP features, but you do not really need to care about this at the moment — only tell the truth and everything should be alright. You've got to trust your supervisors!

Modules

Modules is a list of one element, the name of the callback module used by the child behavior. The exception to that is when you have callback modules whose identity you do not know beforehand (such as event handlers in an event manager). In this case, the value of *Modules* should be dynamic so that the whole OTP system knows who to contact when using more advanced features, such as releases.

update:

Since version 18.0, the supervisor structure can be provided as maps, of the form `{#{strategy => RestartStrategy, intensity => MaxRestart, period => MaxTime}, [#{id => ChildId, start => StartFunc, restart => Restart, shutdown => Shutdown, type => Type, modules => Module}]}.`

This is pretty much the same structure as the existing one, but with maps instead of tuples. The supervisor module defines some default values, but to be clear and readable for people who will maintain your code, having the whole specification explicit is a good idea.

Hooray, we now have the basic knowledge required to start supervised processes. You can take a break and digest it all, or move forward with more content!



Testing it Out

Some practice is in order. And in term of practice, the perfect example is a band practice. Well not that perfect, but bear with me for a while, because we'll go on quite an analogy as a pretext to try our hand at writing supervisors and whatnot.

We're managing a band named *RSYNC, made of programmers playing a few common instruments: a drummer, a singer, a bass player and a keytar player, in memory of all the forgotten 80's glory. Despite a few retro hit song covers such as 'Thread Safety Dance' and 'Saturday Night Coder', the band has a hard time getting a venue. Annoyed with the whole situation, I storm into your office with yet another sugar rush-induced idea of simulating a band in Erlang because "at least we won't be hearing our guys". You're tired because you live in the same apartment as the drummer (who is the weakest link in this band, but they stick together with him because they do not know any other drummer, to be honest), so you accept.

Musicians

The first thing we can do is write the individual band members. For our use case, the musicians module will implement a gen_server. Each musician will take an instrument and a skill level as a parameter (so we can say the drummer sucks, while the others are alright). Once a musician has spawned, it shall start playing. We'll also have an option to stop them, if needed. This gives us the following module and interface:

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

-export([start_link/2, stop/1]).
-export([init/1, handle_call/3, handle_cast/2,
        handle_info/2, code_change/3, terminate/2]).

-record(state, {name = "", role, skill = good}).
-define(DELAY, 750).

start_link(Role, Skill) ->
    gen_server:start_link({local, Role}, ?MODULE, [Role, Skill], []).

stop(Role) -> gen_server:call(Role, stop).
```

I've defined a ?DELAY macro that we'll use as the standard time span between each time a musician will show himself as playing. As the record definition shows, we'll also have to give each of them a name:

```

init([Role, Skill]) ->
    %% To know when the parent shuts down
    process_flag(trap_exit),
    %% sets a seed for random number generation for the life of the process
    %% uses the current time to do it. Unique value guaranteed by now()
    random:seed(now()),
    TimeToPlay = random:uniform(3000),
    Name = pick_name(),
    StrRole = atom_to_list(Role),
    io:format("Musician ~s, playing the ~s entered the room~n",
              [Name, StrRole]),
    {ok, #state{name=Name, role=StrRole, skill=Skill}, TimeToPlay}.

```

Two things go on in the init/1 function. First we start trapping exits. If you recall the description of the terminate/2 from the [Generic Servers chapter](#), we need to do this if we want terminate/2 to be called when the server's parent shuts down its children. The rest of the init/1 function is setting a random seed (so that each process gets different random numbers) and then creates a random name for itself. The functions to create the names are:

```

%% Yes, the names are based off the magic school bus characters'
%% 10 names!
pick_name() ->
    %% the seed must be set for the random functions. Use within the
    %% process that started with init/1
    lists:nth(random:uniform(10), firstnames())
    ++ " "
    lists:nth(random:uniform(10), lastnames()).

firstnames() ->
    ["Valerie", "Arnold", "Carlos", "Dorothy", "Keesha",
     "Phoebe", "Ralphie", "Tim", "Wanda", "Janet"].

lastnames() ->
    ["Frizzle", "Perlstein", "Ramon", "Ann", "Franklin",
     "Terese", "Tennelli", "Jamal", "Li", "Perlstein"].

```

Alright! We can move on to the implementation. This one is going to be pretty trivial for handle_call and handle_cast:

```

handle_call(stop, _From, S=#state{}) ->
    {stop, normal, ok, S};
handle_call(_Message, _From, S) ->
    {noreply, S, ?DELAY}.

```

```
handle_cast(_Message, S) ->
    {noreply, S, ?DELAY}.
```

The only call we have is to stop the musician server, which we agree to do pretty quick. If we receive an unexpected message, we do not reply to it and the caller will crash. Not our problem. We set the timeout in the {noreply, S, ?DELAY} tuples, for one simple reason that we'll see right now:

```
handle_info(timeout, S = #state{name=N, skill=good}) ->
    io:format("~s produced sound!~n",[N]),
    {noreply, S, ?DELAY};
handle_info(timeout, S = #state{name=N, skill=bad}) ->
    case random:uniform(5) of
        1 ->
            io:format(~s played a false note. Uh oh~n,[N]),
            {stop, bad_note, S};
        _ ->
            io:format(~s produced sound!~n,[N]),
            {noreply, S, ?DELAY}
    end;
handle_info(_Message, S) ->
    {noreply, S, ?DELAY}.
```

Each time the server times out, our musicians are going to play a note. If they're good, everything's going to be completely fine. If they're bad, they'll have one chance out of 5 to miss and play a bad note, which will make them crash. Again, we set the ?DELAY timeout at the end of each non-terminating call.

Then we add an empty code_change/3 callback, as required by the 'gen_server' behaviour:

```
code_change(_OldVsn, State, _Extra) ->
    {ok, State}.
```

And we can set the terminate function:

```
terminate(normal, S) ->
    io:format(~s left the room (~s)~n,[S#state.name, S#state.role]);
terminate(bad_note, S) ->
    io:format(~s sucks! kicked that member out of the band! (~s)~n",
              [S#state.name, S#state.role]);
terminate(shutdown, S) ->
    io:format("The manager is mad and fired the whole band! "
```

```

"~s just got back to playing in the subway~n",
[S#state.name]);
terminate(_Reason, S) ->
io:format("~s has been kicked out (~s)~n", [S#state.name, S#state.role]).
```



We've got many different messages here. If we terminate with a normal reason, it means we've called the `stop/1` function and so we display the the musician left of his/her own free will. In the case of a `bad_note` message, the musician will crash and we'll say that it's because the manager (the supervisor we'll soon add) kicked him out of the game.

Then we have the `shutdown` message, which will come from the supervisor. Whenever that happens, it means the supervisor decided to kill all of its children, or in our case, fired all of his musicians. We then add a generic error message for the rest.

Here's a simple use case of a musician:

```

1> c(musicians).
{ok,musicians}
2> musicians:start_link(bass, bad).
Musician Ralphie Franklin, playing the bass entered the room
{ok,<0.615.0>}
Ralphie Franklin produced sound!
Ralphie Franklin produced sound!
Ralphie Franklin played a false note. Uh oh
Ralphie Franklin sucks! kicked that member out of the band! (bass)
3>
=ERROR REPORT==== 6-Mar-2011::03:22:14 ===
** Generic server bass terminating
** Last message in was timeout
** When Server state == {state,"Ralphie Franklin","bass",bad}
** Reason for termination ==
** bad_note
** exception error: bad_note
```

So we have Ralphie playing and crashing after a bad note. Hooray. If you try the same with a good musician, you'll need to call our `musicians:stop(Instrument)` function

in order to stop all the playing.

Band Supervisor

We can now work with the supervisor. We'll have three grades of supervisors: a lenient one, an angry one, and a total jerk. The difference between them is that the lenient supervisor, while still a very pissy person, will fire a single member of the band at a time (`one_for_one`), the one who fails, until he gets fed up, fires them all and gives up on bands. The angry supervisor, on the other hand, will fire some of them (`rest_for_one`) on each mistake and will wait shorter before firing them all and giving up. Then the jerk supervisor will fire the whole band each time someone makes a mistake, and give up if the bands fail even less often.

```
-module(band_supervisor).  
-behaviour(supervisor).
```

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

```
start_link(Type) ->  
    supervisor:start_link({local,?MODULE}, ?MODULE, Type).
```

```
%% The band supervisor will allow its band members to make a few  
%% mistakes before shutting down all operations, based on what  
%% mood he's in. A lenient supervisor will tolerate more mistakes  
%% than an angry supervisor, who'll tolerate more than a  
%% complete jerk supervisor  
init(lenient) ->  
    init({one_for_one, 3, 60});  
init(angry) ->  
    init({rest_for_one, 2, 60});  
init(jerk) ->  
    init({one_for_all, 1, 60});
```

The init definition doesn't finish there, but this lets us set the tone for each of the kinds of supervisor we want. The lenient one will only restart one musician and will fail on the fourth failure in 60 seconds. The second one will accept only 2 failures and the jerk supervisor will have very strict standards there!

Now let's finish the function and actually implement the band starting functions and whatnot:

```

init([RestartStrategy, MaxRestart, MaxTime]) ->
{ok, [{RestartStrategy, MaxRestart, MaxTime},
      [{singer,
        {musicians, start_link, [singer, good]},
        permanent, 1000, worker, [musicians]},
       {bass,
        {musicians, start_link, [bass, good]},
        temporary, 1000, worker, [musicians]},
       {drum,
        {musicians, start_link, [drum, bad]},
        transient, 1000, worker, [musicians]},
       {keytar,
        {musicians, start_link, [keytar, good]},
        transient, 1000, worker, [musicians]}}
      ]]}.

```

So we can see we'll have 3 good musicians: the singer, bass player and keytar player. The drummer is terrible (which makes you pretty mad). The musicians have different *Restarts* (permanent, transient or temporary), so the band could never work without a singer even if the current one left of his own will, but could still play real fine without a bass player, because frankly, who gives a crap about bass players?

That gives us a functional `band_supervisor` module, which we can now try:

```

3> c(band_supervisor).
{ok,band_supervisor}
4> band_supervisor:start_link(lenient).
Musician Carlos Terese, playing the singer entered the room
Musician Janet Terese, playing the bass entered the room
Musician Keesha Ramon, playing the drum entered the room
Musician Janet Ramon, playing the keytar entered the room
{ok,<0.623.0>}
Carlos Terese produced sound!
Janet Terese produced sound!
Keesha Ramon produced sound!
Janet Ramon produced sound!
Carlos Terese produced sound!
Keesha Ramon played a false note. Uh oh
Keesha Ramon sucks! kicked that member out of the band! (drum)
... <snip> ...
Musician Arnold Tennelli, playing the drum entered the room
Arnold Tennelli produced sound!
Carlos Terese produced sound!

```

Janet Terese produced sound!
Janet Ramon produced sound!
Arnold Tennelli played a false note. Uh oh
Arnold Tennelli sucks! kicked that member out of the band! (drum)
... <snip> ...
Musician Carlos Frizzle, playing the drum entered the room
... <snip for a few more firings> ...
Janet Jamal played a false note. Uh oh
Janet Jamal sucks! kicked that member out of the band! (drum)
The manager is mad and fired the whole band! Janet Ramon just got back to playing in the subway
The manager is mad and fired the whole band! Janet Terese just got back to playing in the subway
The manager is mad and fired the whole band! Carlos Terese just got back to playing in the subway
** exception error: shutdown

Magic! We can see that only the drummer is fired, and after a while, everyone gets it too. And off to the subway (tubes for the UK readers) they go!

You can try with other kinds of supervisors and it will end the same. The only difference will be the restart strategy:

5> band_supervisor:start_link(angry).
Musician Dorothy Frizzle, playing the singer entered the room
Musician Arnold Li, playing the bass entered the room
Musician Ralphie Perlstein, playing the drum entered the room
Musician Carlos Perlstein, playing the keytar entered the room
... <snip> ...
Ralphie Perlstein sucks! kicked that member out of the band! (drum)
...
The manager is mad and fired the whole band! Carlos Perlstein just got back to playing in the subway

For the angry one, both the drummer and the keytar players get fired when the drummer makes a mistake. This nothing compared to the jerk's behaviour:

6> band_supervisor:start_link(jerk).
Musician Dorothy Franklin, playing the singer entered the room
Musician Wanda Tennelli, playing the bass entered the room
Musician Tim Perlstein, playing the drum entered the room
Musician Dorothy Frizzle, playing the keytar entered the room
... <snip> ...
Tim Perlstein played a false note. Uh oh
Tim Perlstein sucks! kicked that member out of the band! (drum)
The manager is mad and fired the whole band! Dorothy Franklin just got back to playing in the subway
The manager is mad and fired the whole band! Wanda Tennelli just got back to playing in the subway
The manager is mad and fired the whole band! Dorothy Frizzle just got back to playing in the subway

That's most of it for the restart strategies that are not dynamic.

Dynamic Supervision

So far the kind of supervision we've seen has been static. We specified all the children we'd have right in the source code and let everything run after that. This is how most of your supervisors might end up being set in real world applications; they're usually there for the supervision of architectural components. On the other hand, you have supervisors who act over undetermined workers. They're usually there on a per-demand basis. Think of a web server that would spawn a process per connection it receives. In this case, you would want a dynamic supervisor to look over all the different processes you'll have.

Every time a worker is added to a supervisor using the `one_for_one`, `rest_for_one`, or `one_for_all` strategies, the child specification is added to a list in the supervisor, along with a pid and some other information. The child specification can then be used to restart the child and whatnot. Because things work that way, the following interface exists:

`start_child(SupervisorNameOrPid, ChildSpec)`

This adds a child specification to the list and starts the child with it
`terminate_child(SupervisorNameOrPid, ChildId)`

Terminates or brutal_kills the child. The child specification is left in the supervisor

`restart_child(SupervisorNameOrPid, ChildId)`

Uses the child specification to get things rolling.

`delete_child(SupervisorNameOrPid, ChildId)`

Gets rid of the ChildSpec of the specified child

`check_childspecs([ChildSpec])`

Makes sure a child specification is valid. You can use this to try it before using '`start_child/2`'.

`count_children(SupervisorNameOrPid)`

Counts all the children under the supervisor and gives you a little comparative list of who's active, how many specs there are, how many are supervisors and how many are workers.

`which_children(SupervisorNameOrPid)`

gives you a list of all the children under the supervisor.

Let's see how this works with musicians, with the output removed (you need to be quick to outrace the failing drummer!)

```
1> band_supervisor:start_link(lenient).
{ok,0.709.0}
2> supervisor:which_children(band_supervisor).
[{{keytar,<0.713.0>,worker,[musicians]}, {drum,<0.715.0>,worker,[musicians]}, {bass,<0.711.0>,worker,[musicians]}, {singer,<0.710.0>,worker,[musicians]}}]
3> supervisor:terminate_child(band_supervisor, drum).
ok
4> supervisor:terminate_child(band_supervisor, singer).
ok
5> supervisor:restart_child(band_supervisor, singer).
{ok,<0.730.0>}
6> supervisor:count_children(band_supervisor).
[{{specs,4},{active,3},{supervisors,0},{workers,4}}]
7> supervisor:delete_child(band_supervisor, drum).
ok
8> supervisor:restart_child(band_supervisor, drum).
{error,not_found}
9> supervisor:count_children(band_supervisor).
[{{specs,3},{active,3},{supervisors,0},{workers,3}}]
```

And you can see how you could dynamically manage the children. This works well for anything dynamic which you need to manage (I want to start this one, terminate it, etc.) and which are in little number. Because the internal representation is a list, this won't work very well when you need quick access to many children.



In these case, what you want is simple_one_for_one. The problem with simple_one_for_one is that it will not allow you to manually restart a child, delete it or terminate it. This loss in flexibility is fortunately accompanied by a few

advantages. All the children are held in a dictionary, which makes looking them up fast. There is also a single child specification for all children under the supervisor. This will save you memory and time in that you will never need to delete a child yourself or store any child specification.

For the most part, writing a `simple_one_for_one` supervisor is similar to writing any other type of supervisor, except for one thing. The argument list in the `{M,F,A}` tuple is not the whole thing, but is going to be appended to what you call it with when you do `supervisor:start_child(Sup, Args)`. That's right, `supervisor:start_child/2` changes API. So instead of doing `supervisor:start_child(Sup, Spec)`, which would call `erlang:apply(M,F,A)`, we now have `supervisor:start_child(Sup, Args)`, which calls `erlang:apply(M,F,A++Args)`.

Here's how we'd write it for our `band_supervisor`. Just add the following clause somewhere in it:

```
init(jamband) ->
    {ok, {{simple_one_for_one, 3, 60},
          [{jam_musician,
            {musicians, start_link, []},
            temporary, 1000, worker, [musicians]}}]};
```

I've made them all temporary in this case, and the supervisor is quite lenient:

```
1> supervisor:start_child(band_supervisor, [djembe, good]).  
Musician Janet Tennelli, playing the djembe entered the room  
{ok,<0.690.0>}  
2> supervisor:start_child(band_supervisor, [djembe, good]).  
{error,{already_started,<0.690.0>}}
```

Whoops! this happens because we register the djembe player as `djembe` as part of the start call to our `gen_server`. If we didn't name them or used a different name for each, it wouldn't cause a problem. Really, here's one with the name `drum` instead:

```
3> supervisor:start_child(band_supervisor, [drum, good]).  
Musician Arnold Ramon, playing the drum entered the room  
{ok,<0.696.0>}  
3> supervisor:start_child(band_supervisor, [guitar, good]).  
Musician Wanda Perlstein, playing the guitar entered the room  
{ok,<0.698.0>}  
4> supervisor:terminate_child(band_supervisor, djembe).  
{error,simple_one_for_one}
```

Right. As I said, no way to control children that way.

```
5> musicians:stop(drum).
Arnold Ramon left the room (drum)
ok
```

And this works better.

As a general (and sometimes wrong) hint, I'd tell you to use standard supervisors dynamically only when you know with certainty that you will have few children to supervise and/or that they won't need to be manipulated with any speed and rather infrequently. For other kinds of dynamic supervision, use `simple_one_for_one` where possible.

update:

Since version R14B03, it is possible to terminate children with the function `supervisor:terminate_child(SupRef, Pid)`. Simple one for one supervision schemes are now possible to make fully dynamic and have become an all-around interesting choice for when you have many processes running a single type of process.

That's about it for the supervision strategies and child specification. Right now you might be having doubts on 'how the hell am I going to get a working application out of that?' and if that's the case, you'll be happy to get to the next chapter, which actually builds a simple application with a short supervision tree, to see how it could be done in the real world.

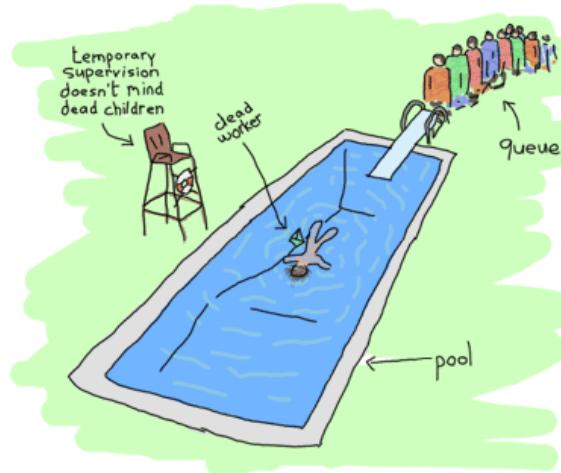
Building an Application With OTP

We've now seen how to use generic servers, finite state machine, event handlers and supervisors. We've not exactly seen how to use them together to build applications and tools, though.

An Erlang application is a group of related code and processes. An *OTP application* specifically uses OTP behaviours for its processes, and then wraps them in a very specific structure that tells the VM how to set everything up and then tear it down.

So in this chapter, we're going to build an application with OTP components, but not a full OTP one because we won't do the whole wrapping up just now. The details of complete OTP applications are a bit complex and warrant their own chapter (the next one). This one's going to be about implementing a process pool. The idea behind such a process pool is to manage and limit resources running in a system in a generic manner.

A Pool of Processes



So yes, a pool allows to limit how many processes run at once. A pool can also queue up jobs when the running workers limit is hit. The jobs can then be ran as soon as resources are freed up or simply block by telling the user they can't do anything else. Despite real world pools doing nothing similar to actual process pools, there are reasons to want to use the latter. A few of them could include:

- Limiting a server to N concurrent connections at most;

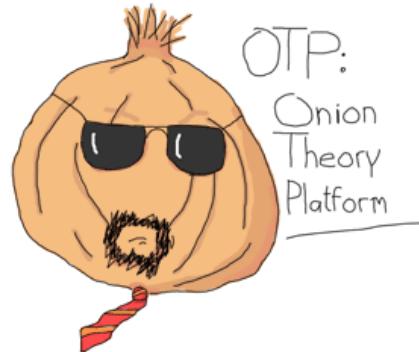
- Limiting how many files can be opened by an application;
- Giving different priorities to different subsystems of a release by allowing more resources for some and less for others. Let's say allowing more processes for client requests than processes in charge of generating reports for management.
- Allowing an application under occasional heavy loads coming in bursts to remain more stable during its entire life by queuing the tasks.

Our process pool application will thus need to support a few functions:

- Starting and stopping the application
- Starting and stopping a particular process pool (all the pools sit within the process pool application)
- Running a task in the pool and telling you it can't be started if the pool is full
- Running a task in the pool if there's room, otherwise keep the calling process waiting while the task is in the queue. Free the caller once the task can be run.
- Running a task asynchronously in the pool, as soon as possible. If no place is available, queue it up and run it whenever.

These needs will help drive our program design. Also keep in mind that we can now use supervisors. And of course we want to use them. The thing is, if they give us new powers in term of robustness, they also impose a certain limit on flexibility. Let's explore that.

The Onion Layer Theory



To help ourselves design an application with supervisors, it helps to have an idea of what needs supervision and how it needs to be supervised. You'll recall we

have different strategies with different settings; these will fit for different kinds of code with different kinds of errors. A rainbow of mistakes can be made!

One thing newcomers and even experienced Erlang programmers have trouble dealing with is usually how to cope with the loss of state. Supervisors kill processes, state is lost, woe is me. To help with this, we will identify different kinds of state:

- Static state. This type can easily be fetched from a config file, another process or the supervisor restarting the application.
- Dynamic state, composed of data you can re-compute. This includes state that you had to transform from its initial form to get where it is right now
- Dynamic data you can not recompute. This might include user input, live data, sequences of external events, etc.

Now, static data is somewhat easy to deal with. Most of the time you can get it straight from the supervisor. Same for the dynamic but re-computable data. In this case you might want to grab it and compute it within the `init/1` function, or anywhere else in your code, really.

The most problematic kind of state is the dynamic data you can't recompute and that you can basically just hope not to lose. In some cases you'll be pushing that data to a database although that won't always be a good option.

The idea of an onion layered system is to allow all of these different states to be protected correctly by isolating different kinds of code from each other. It's process segregation.

The static state can be handled by supervisors, the system being started up, etc. Each time a child dies, the supervisor restarts them and can inject them with some form of static state, always being available. Because most supervisor definitions are rather static by nature, each layer of supervision you add acts as a shield protecting your application against their failure and the loss of their state.

The dynamic state that can be recomputed has a whole lot of available solutions: build it from the static data sent by the supervisors, go fetch it back from some other process, database, text file, the current environment or whatever. It should be relatively easy to get it back on each restart. The fact that

you have supervisors that do a restarting job can be enough to help you keep that state alive.

The dynamic non-recomputable kind of state needs a more thoughtful approach. The real nature of an onion-layered approach takes shape here. The idea is that the most important data (or the hardest to find back) has to be the most protected type. The places where you are actually not allowed to fail is called the *error kernel* of your application.



The error kernel is likely the place where you'll want to use `try ... catches` more than anywhere else, where handling exceptional cases is vital. This is what you want to be error-free. Careful testing has to be done around there, especially in cases where there is no way to go back. You don't want to lose a customer's order halfway through processing it, do you? Some operations are going to be considered safer than others. Because of this, we want to keep vital data in the safest core possible, and keeping everything somewhat dangerous outside of it. In specific terms, this means that all kinds of operations related together should be part of the same supervision trees, and the unrelated ones should be kept in different trees. Within the same tree, operations that are failure-prone but not vital can be in a separate sub-tree. When possible, only restart the part of the tree that needs it. We'll see an example of this when designing our actual process pool's supervision tree.

A Pool's Tree

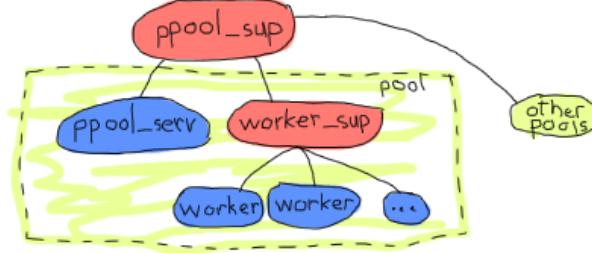
So how should we organise these process pools? There are two schools of thought here. One tells people to design bottom-up (write all individual components, put them together as required) and another one tells us to write things top-down (design as if all the parts were there, then build them). Both approaches are equally valid depending on the circumstances and your personal style. For the sake of making things understandable, we're going to do things top-down here.

So what should our tree look like? Well our requirements include: being able to start the pool application as a whole, having many pools and each pool having many workers that can be queued. This already suggests a few possible design constraints.

We will need one gen_server per pool. The server's job will be to maintain the counter of how many workers are in the pool. For convenience, the same server should also hold the queue of tasks. Who should be in charge of overlooking each of the workers, though? The server itself?

Doing it with the server is interesting. After all, it needs to track the processes to count them and supervising them itself is a nifty way to do it. Moreover neither the server nor the processes can crash without losing the state of all the others (otherwise the server can't track the tasks after it restarted). It has a few disadvantages too: the server has many responsibilities, can be seen as more fragile and duplicates the functionality of existing, better tested modules.

A good way to make sure all workers are properly accounted for would be to use a supervisor just for them



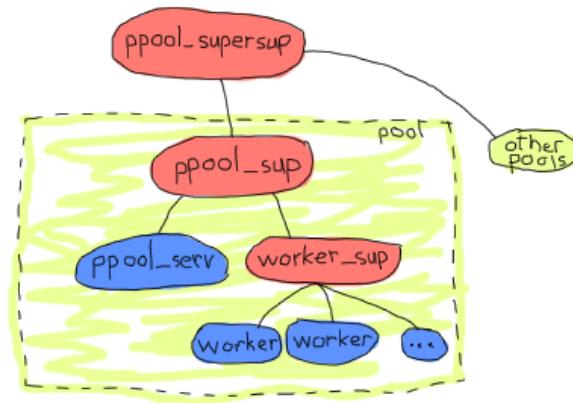
The one above, for example would have a single supervisor for all of the pools. Each pool is in fact a set of a pool server and a supervisor for workers. The pool server knows the existence of its worker supervisor and asks it to add items. Given adding children is a very dynamic thing with unknown limits so far, a simple_one_for_one supervisor shall be used.

Note: the name ppool is chosen because the Erlang standard library already has a pool module. Plus it's a terrible pool-related pun.

The advantage of doing things that way is that because the worker_sup supervisor will need to track only OTP workers of a single type, each pool is guaranteed to be about a well defined kind of worker, with simple management and restart

strategies that are easy to define. This right here is one example of an error kernel being better defined. If I'm using a pool of sockets for web connections and another pool of servers in charge of log files, I am making sure that incorrect code or messy permissions in the log file section of my application won't be drowning out the processes in charge of the sockets. If the log files' pool crashes too much, they'll be shut down and their supervisor will stop. Oh wait!

Right. Because all pools are under the same supervisor, a given pool or server restarting too many times in a short time span can take all the other pools down. This means what we might want to do is add one level of supervision. This will also make it much simpler to handle more than one pool at a time, so let's say the following will be our application architecture:



And that makes a bit more sense. From the onion layer perspective, all pools are independent, the workers are independent from each other and the ppool_serv server is going to be isolated from all the workers. That's good enough for the architecture, everything we need seems to be there. We can start working on the implementation, again, top to bottom.

Implementing the Supervisors

We can start with just the top level supervisor, ppool_supersup. All this one has to do is start the supervisor of a pool when required. We'll give it a few functions: start_link/0, which starts the whole application, stop/0, which stops it, start_pool/3, which creates a specific pool and stop_pool/1 which does the opposite. We also can't forget init/1, the only callback required by the supervisor behaviour:

```
-module(ppool_supersup).
-behaviour(supervisor).
```

```
-export([start_link/0, stop/0, start_pool/3, stop_pool/1]).  
-export([init/1]).
```

```
start_link() ->  
    supervisor:start_link({local, ppool}, ?MODULE, []).
```

Here we gave the top level process pool supervisor the name `ppool` (this explains the use of `{local, Name}`, an OTP convention about registering `gen_*` processes on a node; another one exists for distributed registration). This is because we know we will only have one `ppool` per Erlang node and we can give it a name without worrying about clashes. Fortunately, the same name can then be used to stop the whole set of pools:

```
%% technically, a supervisor can not be killed in an easy way.
```

```
%% Let's do it brutally!
```

```
stop() ->  
    case whereis(ppool) of  
        P when is_pid(P) ->  
            exit(P, kill);  
        _ -> ok  
    end.
```

As the comments in the code explain it, we can not terminate a supervisor gracefully. The reason for this is that the OTP framework provides a well-defined shutdown procedure for all supervisors, but we can't use it from where we are right now. We'll see how to do it in the next chapter, but for now, brutally killing the supervisor is the best we can do.

What is the top level supervisor exactly? Well its only task is to hold pools in memory and supervise them. In this case, it will be a childless supervisor:

```
init([]) ->  
    MaxRestart = 6,  
    MaxTime = 3600,  
    {ok, {{one_for_one, MaxRestart, MaxTime}, []}}.
```

We can now focus on starting each individual pool's supervisor and attaching them to `ppool`. Given our initial requirements, we can determine that we'll need two parameters: the number of workers the pool will accept, and the `{M,F,A}` tuple that the worker supervisor will need to start each worker. We'll also add a name for good measure. We then pass this `childspec` to the process pool's supervisor as we start it:

```

start_pool(Name, Limit, MFA) ->
    ChildSpec = {Name,
        {ppool_sup, start_link, [Name, Limit, MFA]},
        permanent, 10500, supervisor, [ppool_sup]},
    supervisor:start_child(ppool, ChildSpec).

```

You can see each pool supervisor is asked to be permanent, has the arguments needed (notice how we're changing programmer-submitted data into static data this way). The name of the pool is both passed to the supervisor and used as an identifier in the child specification. There's also a maximum shutdown time of 10500. There is no easy way to pick this value. Just make sure it's large enough that all the children will have time to stop. Play with them according to your needs and test and adapt yourself. You might as well try the infinity option if you just don't know.

To stop the pool, we need to ask the ppool super supervisor (the *supersup!*) to kill its matching child:

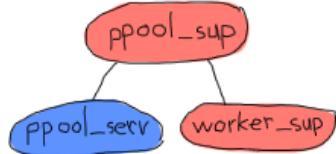
```

stop_pool(Name) ->
    supervisor:terminate_child(ppool, Name),
    supervisor:delete_child(ppool, Name).

```

This is possible because we gave the pool's *Name* as the *childspec* identifier. Great! We can now focus on each pool's direct supervisor!

Each ppool_sup will be in charge of the pool server and the worker supervisor.



Can you see the funny thing here? The ppool_serv process should be able to contact the worker_sup process. If we're to have them started by the same supervisor at the same time, we won't have any way to let ppool_serv know about worker_sup, unless we were to do some trickery with supervisor:which_children/1 (which would be sensitive to timing and somewhat risky), or giving a name to both the ppool_serv process (so that users can call it) and the supervisor. Now we don't want to give names to the supervisors because:

1. The users don't need to call them directly

2. We would need to dynamically generate atoms and that makes me nervous
3. There is a better way.

The way to do it is basically to get the pool server to dynamically attach the worker supervisor to its ppool_sup. If this is vague, you'll get it soon. For now we only start the server:

```
-module(ppool_sup).
-export([start_link/3, init/1]).
-behaviour(supervisor).

start_link(Name, Limit, MFA) ->
    supervisor:start_link(?MODULE, {Name, Limit, MFA}).

init({Name, Limit, MFA}) ->
    MaxRestart = 1,
    MaxTime = 3600,
    {ok, [{one_for_all, MaxRestart, MaxTime},
          [{serv,
            {ppool_serv, start_link, [Name, Limit, self(), MFA]},
            permanent,
            5000, % Shutdown time
            worker,
            [ppool_serv]}]}]}.
```

And that's about it. Note that the *Name* is passed to the server, along with `self()`, the supervisor's own pid. This will let the server call for the spawning of the worker supervisor; the *MFA* variable will be used in that call to let the simple_one_for_one supervisor know what kind of workers to run.

We'll get to how the server handles everything, but for now we'll finish writing all of the application's supervisors by writing `ppool_worker_sup`, in charge of all the workers:

```
-module(ppool_worker_sup).
-export([start_link/1, init/1]).
-behaviour(supervisor).

start_link(MFA = {_, _, _}) ->
    supervisor:start_link(?MODULE, MFA).

init({M,F,A}) ->
    MaxRestart = 5,
    MaxTime = 3600,
```

```
{ok, {{simple_one_for_one, MaxRestart, MaxTime},
      [{ppool_worker,
        {M,F,A},
        temporary, 5000, worker, [M]}]}}.
```

Simple stuff there. We picked a simple_one_for_one because workers could be added in very high number with a requirement for speed, plus we want to restrict their type. All the workers are temporary, and because we use an {M,F,A} tuple to start the worker, we can use any kind of OTP behaviour there.



The reason to make the workers temporary is twofold. First of all, we can not know for sure whether they need to be restarted or not in case of failure or what kind of restart strategy would be required for them. Secondly, the pool might only be useful if the worker's creator can have an access to the worker's pid, depending on the use case. For this to work in any safe and simple manner, we can't just restart workers as we please without tracking its creator and sending it a notification. This would make things quite complex just to grab a pid. Of course, you are free to write your own ppool_worker_sup that doesn't return pids but restarts them. There's nothing inherently wrong in that design.

Working on the Workers

The pool server is the most complex part of the application, where all the clever business logic happens. Here's a reminder of the operations we must support.

- Running a task in the pool and telling you it can't be started if the pool is full
- Running a task in the pool if there's place, otherwise keep the calling process waiting while the task is in the queue, until it can be run.

- Running a task asynchronously in the pool, as soon as possible. If no place is available, queue it up and run it whenever.

The first one will be done by a function named `run/2`, the second by `sync_queue/2` and the last one by `async_queue/2`:

```
-module(ppool_serv).
-behaviour(gen_server).
-export([start/4, start_link/4, run/2, sync_queue/2, async_queue/2, stop/1]).
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
        code_change/3, terminate/2]).

start(Name, Limit, Sup, MFA) when is_atom(Name), is_integer(Limit) ->
    gen_server:start({local, Name}, ?MODULE, {Limit, MFA, Sup}, []).

start_link(Name, Limit, Sup, MFA) when is_atom(Name), is_integer(Limit) ->
    gen_server:start_link({local, Name}, ?MODULE, {Limit, MFA, Sup}, []).

run(Name, Args) ->
    gen_server:call(Name, {run, Args}).

sync_queue(Name, Args) ->
    gen_server:call(Name, {sync, Args}, infinity).

async_queue(Name, Args) ->
    gen_server:cast(Name, {async, Args}).

stop(Name) ->
    gen_server:call(Name, stop).
```

For `start/4` and `start_link/4`, `Args` are going to be the additional arguments passed to the A part of the `{M,F,A}` triple sent to the supervisor. Note that for the synchronous queue, I've set the waiting time to infinity.

As mentioned earlier, we have to start the supervisor from within the server. If you're adding the code as we go, you might want to include an empty `gen_server` template (or use the completed file) to follow along, because we'll do things on a per-feature basis rather than just reading the server from top to bottom.

The first thing we do is handle the creation of the supervisor. If you remember last chapter's bit on [dynamic supervision](#), we do not need a `simple_one_for_one` for

cases where we need few children added, so supervisor:start_child/2 ought to do it. We'll first define the child specification of the worker supervisor:

```
%% The friendly supervisor is started dynamically!
-define(SPEC(MFA),
        {worker_sup,
         {ppool_worker_sup, start_link, [MFA]},
         temporary,
         10000,
         supervisor,
         [ppool_worker_sup]}).
```

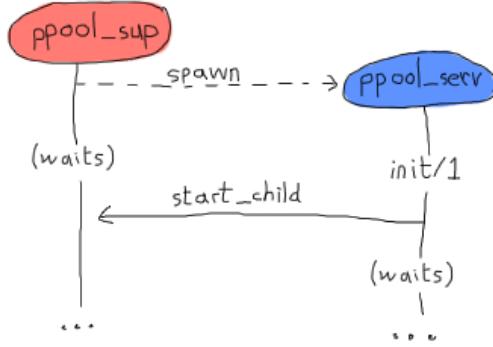
Nothing too special there. We can then define the inner state of the server. We know we will have to track a few pieces of data: the number of process that can be running, the pid of the supervisor and a queue for all the jobs. To know when a worker's done running and to fetch one from the queue to start it, we will need to track each worker from the server. The sane way to do this is with monitors, so we'll also add a refs field to our state record to keep all the monitor references in memory:

```
-record(state, {limit=0,
               sup,
               refs,
               queue=queue:new()}).
```

With this ready, we can start implementing the init function. The natural thing to try is the following:

```
init({Limit, MFA, Sup}) ->
    {ok, Pid} = supervisor:start_child(Sup, ?SPEC(MFA)),
    link(Pid),
    {ok, #state{limit=Limit, refs=gb_sets:empty()}}.
```

and get going. However, this code is wrong. The way things work with gen_* behaviours is that the process that spawns the behaviour waits until the init/1 function returns before resuming its processing. This means that by calling supervisor:start_child/2 in there, we create the following deadlock:



Both processes will keep waiting for each other until there is a crash. The cleanest way to get around this is to create a special message that the server will send to itself to be able to handle it in `handle_info/2` as soon as it has returned (and the pool supervisor has become free):

```

init([{Limit, MFA, Sup}]) ->
    %% We need to find the Pid of the worker supervisor from here,
    %% but alas, this would be calling the supervisor while it waits for us!
    self() ! {start_worker_supervisor, Sup, MFA},
    {ok, #state{limit=Limit, refs=gb_sets:empty()}}.

```

This one is cleaner. We can then head out to the `handle_info/2` function and add the following clauses:

```

handle_info({start_worker_supervisor, Sup, MFA}, S = #state{}) ->
    {ok, Pid} = supervisor:start_child(Sup, ?SPEC(MFA)),
    link(Pid),
    {noreply, S#state{sup=Pid}};
handle_info(Msg, State) ->
    io:format("Unknown msg: ~p~n", [Msg]),
    {noreply, State}.

```

The first clause is the interesting one here. We find the message we sent ourselves (which will necessarily be the first one received), ask the pool supervisor to add the worker supervisor, track this Pid and voilà! Our tree is now fully initialized. Whew. You can try compiling everything to make sure no mistake has been made so far. Unfortunately we still can't test the application because too much stuff is missing.

Note: Don't worry if you do not like the idea of building the whole application before running it. Things are being done this way to show a cleaner reasoning of the whole thing. While I did have the general design in mind (the same one I

illustrated earlier), I started writing this pool application in a little test-driven manner with a few tests here and there and a bunch of refactorings to get everything to a functional state.

Few Erlang programmers (much like programmers of most other languages) will be able to produce production-ready code on their first try, and the author is not as clever as the examples might make it look like.

Alright, so we've got this bit solved. Now we'll take care of the `run/2` function. This one is a synchronous call with the message of the form `{run, Args}` and works as follows:

```
handle_call({run, Args}, _From, S = #state{limit=N, sup=Sup, refs=R}) when N > 0 ->
    {ok, Pid} = supervisor:start_child(Sup, Args),
    Ref = erlang:monitor(process, Pid),
    {reply, {ok,Pid}, S#state{limit=N-1, refs=gb_sets:add(Ref,R)}};
handle_call({run, _Args}, _From, S=#state{limit=N}) when N =< 0 ->
    {reply, noalloc, S};
```

A long function head, but we can see most of the management taking place there. Whenever there are places left in the pool (the original limit N being decided by the programmer adding the pool in the first place), we accept to start the worker. We then set up a monitor to know when it's done, store all of this in our state, decrement the counter and off we go.

In the case no space is available, we simply reply with `noalloc`.

The calls to `sync_queue/2` will give a very similar implementation:

```
handle_call({sync, Args}, _From, S = #state{limit=N, sup=Sup, refs=R}) when N > 0 ->
    {ok, Pid} = supervisor:start_child(Sup, Args),
    Ref = erlang:monitor(process, Pid),
    {reply, {ok,Pid}, S#state{limit=N-1, refs=gb_sets:add(Ref,R)}};
handle_call({sync, Args}, From, S = #state{queue=Q}) ->
    {noreply, S#state{queue=queue:in({From, Args}, Q)}};
```

If there is space for more workers, then the first clause is going to do exactly the same as we did for `run/2`. The difference comes in the case where no workers can run. Rather than replying with `noalloc` as we did last time, this one doesn't reply to the caller, keeps the `From` information and enqueues it for a later time when there is space for the worker to be run. We'll see how we dequeue them and

handle them soon enough, but for now, we'll finish the handle_call/3 callback with the following clauses:

```
handle_call(stop, _From, State) ->
    {stop, normal, ok, State};
handle_call(_Msg, _From, State) ->
    {noreply, State}.
```

Which handle the unknown cases and the stop/1 call. We can now focus on getting async_queue/2 working. Because async_queue/2 basically does not care when the worker is ran and expects absolutely no reply, it was decided to make it a cast rather than a call. You'll find the logic of it to be awfully similar to the two previous options:

```
handle_cast({async, Args}, S=#state{limit=N, sup=Sup, refs=R}) when N > 0 ->
    {ok, Pid} = supervisor:start_child(Sup, Args),
    Ref = erlang:monitor(process, Pid),
    {noreply, S#state{limit=N-1, refs=gb_sets:add(Ref,R)}};
handle_cast({async, Args}, S=#state{limit=N, queue=Q}) when N < 0 ->
    {noreply, S#state{queue=queue:in(Args,Q)}};
%% Not going to explain this one!
handle_cast(_Msg, State) ->
    {noreply, State}.
```

Again, the only big difference apart from not replying is that when there is no place left for a worker it is queued. This time though, we have no *From* information and just send it to the queue without it; the limit doesn't change in this case.

When do we know it's time to dequeue something? Well, we have monitors set all around the place and we're storing their references in a gb_sets. Whenever a worker goes down, we're notified of it. Let's work from there:

```
handle_info({'DOWN', Ref, process, _Pid, _}, S = #state{refs=Refs}) ->
    io:format("received down msg~n"),
    case gb_sets:is_element(Ref, Refs) of
        true ->
            handle_down_worker(Ref, S);
        false -> %% Not our responsibility
            {noreply, S}
    end;
handle_info({start_worker_supervisor, Sup, MFA}, S = #state{}) ->
    ...
```

```
handle_info(Msg, State) ->
```

```
...
```

What we do in the snippet is make sure the 'DOWN' message we get comes from a worker. If it doesn't come from one (which would be surprising), we just ignore it. However, if the message really is what we want, we call a function named `handle_down_worker/2`:

```
handle_down_worker(Ref, S = #state{limit=L, sup=Sup, refs=Refs}) ->
    case queue:out(S#state.queue) of
        {{value, {From, Args}}, Q} ->
            {ok, Pid} = supervisor:start_child(Sup, Args),
            NewRef = erlang:monitor(process, Pid),
            NewRefs = gb_sets:insert(NewRef, gb_sets:delete(Ref, Refs)),
            gen_server:reply(From, {ok, Pid}),
            {noreply, S#state{refs=NewRefs, queue=Q}};
        {{value, Args}, Q} ->
            {ok, Pid} = supervisor:start_child(Sup, Args),
            NewRef = erlang:monitor(process, Pid),
            NewRefs = gb_sets:insert(NewRef, gb_sets:delete(Ref, Refs)),
            {noreply, S#state{refs=NewRefs, queue=Q}};
        {empty, _} ->
            {noreply, S#state{limit=L+1, refs=gb_sets:delete(Ref, Refs)}}
    end.
```

Quite a complex one. Because our worker is dead, we can look in the queue for the next one to run. We do this by popping one element out of the queue, and looking what the result is. If there is at least one element in the queue, it will be of the form `{{value, Item}, NewQueue}`. If the queue is empty, it returns `{empty, SameQueue}`. Furthermore, we know that when we have the value `{From, Args}`, it means this came from `sync_queue/2` and that it came from `async_queue/2` otherwise.

Both cases where the queue has tasks in it will behave roughly the same: a new worker is attached to the worker supervisor, the reference of the old worker's monitor is removed and replaced with the new worker's monitor reference. The only different aspect is that in the case of the synchronous call, we send a manual reply while in the other we can remain silent. That's about it.

In the case the queue was empty, we need to do nothing but increment the worker limit by one.

The last thing to do is add the standard OTP callbacks:

```
code_change(_OldVsn, State, _Extra) ->
    {ok, State}.
```

```
terminate(_Reason, _State) ->
    ok.
```

That's it, our pool is ready to be used! It is a very unfriendly pool, though. All the functions we need to use are scattered around the place. Some are in `ppool_supersup`, some are in `ppool_serv`. Plus the module names are long for no reason. To make things nicer, add the following API module (just abstracting the calls away) to the application's directory:

```
%% API module for the pool
-module(ppool).
-export([start_link/0, stop/0, start_pool/3,
        run/2, sync_queue/2, async_queue/2, stop_pool/1]).
```

```
start_link() ->
    ppool_supersup:start_link().
```

```
stop() ->
    ppool_supersup:stop().
```

```
start_pool(Name, Limit, {M,F,A}) ->
    ppool_supersup:start_pool(Name, Limit, {M,F,A}).
```

```
stop_pool(Name) ->
    ppool_supersup:stop_pool(Name).
```

```
run(Name, Args) ->
    ppool_serv:run(Name, Args).
```

```
async_queue(Name, Args) ->
    ppool_serv:async_queue(Name, Args).
```

```
sync_queue(Name, Args) ->
    ppool_serv:sync_queue(Name, Args).
```

And now we're done for real!

Note: you'll have noticed that our process pool doesn't limit the number of items that can be stored in the queue. In some cases, a real server application will need to put a ceiling on how many things can be queued to avoid crashing when too much memory is used, although the problem can be circumvented if you

only use `run/2` and `sync_queue/2` with a fixed number of callers (if all content producers are stuck waiting for free space in the pool, they stop producing so much content in the first place).

Adding a limit to the queue size is left as an exercise to the reader, but fear not because it is relatively simple to do; you will need to pass a new parameter to all functions up to the server, which will then check the limit before any queuing.

Additionally, to control the load of your system, you sometimes want to impose limits closer to their source by using synchronous calls. Synchronous calls allow to block incoming queries when the system is getting swamped by producers faster than consumers; this generally helps keep it more responsive than a free-for-all load.

Writing a Worker

Look at me go, I'm lying all the time! The pool isn't really ready to be used. We don't have a worker at the moment. I forgot. This is a shame because we all know that in the [chapter about writing a concurrent application](#), we've written ourselves a nice task reminder. It apparently wasn't enough for me, so for this one right here, I'll have us writing a *nagger*.

It will basically be a worker for each task, and the worker will keep nagging us by sending repeated messages until a given deadline. It'll be able to take:

- a time delay for which to nag,
- an address (pid) to say where the messages should be sent
- a nagging message to be sent in the process mailbox, including the nagger's own pid to be able to call...
- ... a stop function to say the task is done and that the nagger can stop nagging

Here we go:

```
%% demo module, a nagger for tasks,  
%% because the previous one wasn't good enough  
-module(ppool_nagger).  
-behaviour(gen_server).  
-export([start_link/4, stop/1]).  
-export([init/1, handle_call/3, handle_cast/2,  
        handle_info/2, code_change/3, terminate/2]).
```

```
start_link(Task, Delay, Max, SendTo) ->
    gen_server:start_link(?MODULE, {Task, Delay, Max, SendTo} , []).
```

```
stop(Pid) ->
    gen_server:call(Pid, stop).
```

Yes, we're going to be using yet another `gen_server`. You'll find out that people use them all the time, even when sometimes not appropriate! It's important to remember that our pool can accept any OTP compliant process, not just `gen_servers`.

```
init({Task, Delay, Max, SendTo}) ->
    {ok, {Task, Delay, Max, SendTo}, Delay}.
```

This just takes the basic data and forwards it. Again, `Task` is the thing to send as a message, `Delay` is the time spent in between each sending, `Max` is the number of times it's going to be sent and `SendTo` is a pid or a name where the message will go. Note that `Delay` is passed as a third element of the tuple, which means timeout will be sent to `handle_info/2` after `Delay` milliseconds.

Given our API above, most of the server is rather straightforward:

```
%%% OTP Callbacks
handle_call(stop, _From, State) ->
    {stop, normal, ok, State};
handle_call(_Msg, _From, State) ->
    {noreply, State}.

handle_cast(_Msg, State) ->
    {noreply, State}.

handle_info(timeout, {Task, Delay, Max, SendTo}) ->
    SendTo ! {self(), Task},
    if Max =:= infinity ->
        {noreply, {Task, Delay, Max, SendTo}, Delay};
    Max =< 1 ->
        {stop, normal, {Task, Delay, 0, SendTo}};
    Max > 1 ->
        {noreply, {Task, Delay, Max-1, SendTo}, Delay}
    end.
%% We cannot use handle_info below: if that ever happens,
%% we cancel the timeouts (Delay) and basically zombify
%% the entire process. It's better to crash in this case.
%% handle_info(_Msg, State) ->
```

```

%% {noreply, State}.

code_change(_OldVsn, State, _Extra) ->
{ok, State}.

terminate(_Reason, _State) -> ok.

```

The only somewhat complex part here lies in the `handle_info/2` function. As seen back in the [gen_server chapter](#), every time a timeout is hit (in this case, after *Delay* milliseconds), the timeout message is sent to the process. Based on this, we check how many nags were sent to know if we have to send more or just quit. With this worker done, we can actually try this process pool!

Run Pool Run

We can now play with the pool compile all the files and start the pool top-level supervisor itself:

```

$ erlc *.erl
$ erl
Erlang R14B02 (erts-5.8.3) [source] [64-bit] [smp:4:4] [rq:4] [async-threads:0] [hipe] [kernel-poll:false]

Eshell V5.8.3 (abort with ^G)
1> ppool:start_link().
{ok,<0.33.0>}

```

From this point, we can try a bunch of different features of the nagger as a pool:

```

2> ppool:start_pool(nagger, 2, {ppool_nagger, start_link, []}).
{ok,<0.35.0>}
3> ppool:run(nagger, ["finish the chapter!", 10000, 10, self()]). 
{ok,<0.39.0>}
4> ppool:run(nagger, ["Watch a good movie", 10000, 10, self()]). 
{ok,<0.41.0>}
5> flush().
Shell got {<0.39.0>,"finish the chapter!"}
Shell got {<0.39.0>,"finish the chapter!"}
ok
6> ppool:run(nagger, ["clean up a bit", 10000, 10, self()]). 
noalloc
7> flush().
Shell got {<0.41.0>,"Watch a good movie"}
Shell got {<0.39.0>,"finish the chapter!"}
Shell got {<0.41.0>,"Watch a good movie"}
Shell got {<0.39.0>,"finish the chapter!"}

```

```
Shell got {<0.41.0>,"Watch a good movie"}
```

```
...
```

Everything seems to work rather well for the synchronous non-queued runs. The pool is started, tasks are added and messages are sent to the right destination. When we try to run more tasks than allowed, allocation is denied to us. No time for cleaning up, sorry! The others still run fine though.

Note: the ppool is started with start_link/0. If at any time you make an error in the shell, you take down the whole pool and have to start over again. This issue will be addressed in the next chapter.

Note: of course a cleaner nagger would probably call an event manager used to forward messages correctly to all appropriate media. In practice though, many products, protocols and libraries are prone to change and I always hated books that are no longer good to read once external dependencies have passed their time. As such, I tend to keep all external dependencies rather low, if not entirely absent from this tutorial.

We can try the queuing facilities (asynchronous), just to see:

```
8> ppool:async_queue(nagger, ["Pay the bills", 30000, 1, self()]).
```

```
ok
```

```
9> ppool:async_queue(nagger, ["Take a shower", 30000, 1, self()]).
```

```
ok
```

```
10> ppool:async_queue(nagger, ["Plant a tree", 30000, 1, self()]).
```

```
ok
```

```
<wait a bit>
```

```
received down msg
```

```
received down msg
```

```
11> flush().
```

```
Shell got {<0.70.0>,"Pay the bills"}
```

```
Shell got {<0.72.0>,"Take a shower"}
```

```
<wait some more>
```

```
received down msg
```

```
12> flush().
```

```
Shell got {<0.74.0>,"Plant a tree"}
```

```
ok
```

Great! So the queuing works. The log here doesn't show everything in a very clear manner, but what happens there is that the two first nappers run as soon as possible. Then, the worker limit is hit and we need to queue the third one

(planting a tree). When the nags for paying the bills are done for, the tree nagger is scheduled and sends the message a bit later.

The synchronous one will behave differently:

```
13> ppool:sync_queue(nagger, ["Pet a dog", 20000, 1, self()]).  
{ok,<0.108.0>}  
14> ppool:sync_queue(nagger, ["Make some noise", 20000, 1, self()]).  
{ok,<0.110.0>}  
15> ppool:sync_queue(nagger, ["Chase a tornado", 20000, 1, self()]).  
received down msg  
{ok,<0.112.0>}  
received down msg  
16> flush().  
Shell got {<0.108.0>,"Pet a dog"}  
Shell got {<0.110.0>,"Make some noise"}  
ok  
received down msg  
17> flush().  
Shell got {<0.112.0>,"Chase a tornado"}  
ok
```

Again, the log isn't as clear as if you tried it yourself (which I encourage). The basic sequence of events is that two workers are added to the pool. They aren't done running and when we try to add a third one, the shell gets locked up until `ppool_serv` (under the process name `nagger`) receives a worker's down message (`received down msg`). After this, our call to `sync_queue/2` can return and give us the pid of our brand new worker.

We can now get rid of the pool as a whole:

```
18> ppool:stop_pool(nagger).  
ok  
19> ppool:stop().  
** exception exit: killed
```

All pools will be terminated if you decide to just call `ppool:stop()`, but you'll receive a bunch of error messages. This is because we brutally kill the `ppool_supersup` process rather than taking it down correctly (which in turns crashes all child pools), but next chapter will cover how to do that cleanly.

Cleaning the Pool



Looking back on everything, we've managed to write a process pool to do some resource allocation in a somewhat simple manner. Everything can be handled in parallel, can be limited, and can be called from other processes. Pieces of your application that crash can, with the help of supervisors, be replaced transparently without breaking the entirety of it. Once the pool application was ready, we even rewrote a surprisingly large part of our reminder app with very little code.

Failure isolation for a single computer has been taken into account, concurrency is handled, and we now have enough architectural blocks to write some pretty solid server-side software, even though we still haven't really seen good ways to run them from the shell...

The next chapter will show how to package the ppool application into a real OTP application, ready to be shipped and used by other products. So far we haven't seen all the advanced features of OTP, but I can tell you that you're now on a level where you should be able to understand most intermediate to early advanced discussions on OTP and Erlang (the non-distributed part, at least). That's pretty good!

Building OTP Applications

Why Would I Want That?



After seeing our whole application's supervision tree start at once with a simple function call, we might wonder why we would want to make things more complicated than they already are. The concepts behind supervision trees are a bit complex and I could see myself just starting all of these trees and subtrees manually with a script when the system is first set up. Then after that, I would be free to go outside and try to find clouds that look like animals for the rest of the afternoon.

This is entirely true, yes. This is an acceptable way to do things (especially the part about clouds, because these days everything is about cloud computing). However, as for most abstractions made by programmers and engineers, OTP applications are the result of many ad-hoc systems being generalised and made clean. If you were to make an array of scripts and commands to start your supervision trees as described above, and that other developers you work with had

their own, you'd quickly run into massive issues. Then someone would ask something like "Wouldn't it be nice if everyone used the same kind of system to start everything? And wouldn't it even be nicer if they all had the same kind of application structure?"

OTP applications attempt to solve this exact type of problem. They give a directory structure, a way to handle configurations, a way to handle dependencies, create environment variables and configuration, ways to start and stop applications, and a lot of safe control in detecting conflicts and handling live upgrades them without shutting your applications down.

So unless you don't want these aspects (nor the niceties they give, like consistent structures and tools developed for it), this chapter should be of some interest to you.

My Other Car is a Pool

We're going to reuse the ppool application we wrote for last chapter and turn it into a real OTP application.

The first step in doing so is to copy all the ppool related files into a neat directory structure:

```
ebin/  
include/  
priv/  
src/  
- ppool.erl  
- ppool_sup.erl  
- ppool_supersup.erl  
- ppool_worker_sup.erl  
- ppool_serv.erl  
- ppool_nagger.erl
```

```
test/  
- ppool_tests.erl
```

Most directories will for now remain empty. As explained in the [Designing a Concurrent Application](#) chapter, the `ebin/` directory will hold compiled files, the `include/` directory will contain Erlang header (`.hrl`) files, `priv/` will hold executables, other programs, and various specific files needed for the application to work and `src/` will hold the Erlang source files you will need.



You'll note that I added a `test/` directory just for the test file I had before. The reason for this is that tests are somewhat common, but you don't necessarily want them distributed as part of your application – you just need them when developing your code and justifying yourself to your manager ("tests pass, I don't understand why the app killed people"). Other directories like that end up being added as required, depending on the case. One example is the `doc/` directory, added whenever you have EDoc documentation to add to your application.

The four basic directories to have are `ebin/`, `include/`, `priv/` and `src/` and they'll be common to pretty much every OTP application you

get, although only `ebin/` and `priv/` are going to be exported when real OTP systems are deployed.

The Application Resource File

Where do we go from here? Well the first thing to do is to add an application file. This file will tell the Erlang VM what the application is, where it begins and where it ends. This file lives on in the `ebin/` directory, along with all the compiled modules.

This file is usually named `<yourapp>.app` (in our case `ppool.app`) and contains a bunch of Erlang terms defining the application in terms the VM can understand (the VM is pretty bad at guessing stuff!)

Note: some people prefer to keep this file outside of `ebin/` and instead have a file named `<myapp>.app.src` as part of `src/`. Whatever build system they use then copies this file over to `ebin/` or even generates one in order to keep everything clean.

The basic structure of the application file is simply:

{application, ApplicationName, Properties}.

Where *ApplicationName* is an atom and *Properties* is a list of {Key, value} tuples describing the application. They're used by OTP to figure out what your application does and whatnot, they're all optional, but might always be useful to carry around and necessary for some tools. In fact, we'll only look at a subset of them for now and introduce the others as we need them:

{description, "Some description of your application"}

This gives the system a short description of what the application is. The field is optional and defaults to an empty string. I would suggest always defining a description, if only because it makes things easier to read.

{vsn, "1.2.3"}

Tells what's the version of your application. The string takes any format you want. It's usually a good idea to stick to a scheme of the form <major>.<minor>.<patch> or something like that. When we get to tools to help with upgrades and downgrades, the string is used to identify your application's version.

{modules, ModuleList}

Contains a list of all the modules that your application introduces to the system. A module always belongs to at most one application and can not be present in two applications' app files at once. This list lets the system and tools look at dependencies of your application, making sure everything is where it needs to be and that you have no conflicts with other applications already loaded in the system. If you're using a standard OTP structure and are using a build tool like *rebar3*, this is handled for you.

{registered, AtomList}

Contains a list of all the names registered by the application. This lets OTP know when there will be name clashes when you try to bundle a bunch of applications together, but is entirely based on trusting the developers to give good data. We all know this isn't always the case, so blind faith shouldn't be used in this case.

{env, [{Key, Val}]}{}

This is a list of key/values that can be used as a configuration for your application. They can be obtained at run time by calling `application:get_env(Key)` or `application:get_env(AppName, Key)`. The first one will try to find the value in the application file of whatever application you are in at the moment of the call, the second allows you to specify an application in particular. This stuff can be overwritten as required (either at boot time or by using `application:set_env/3-4`.

All in all this is a pretty useful place to store configuration data rather than having a bunch of config files to read in whatever format, without really knowing where to store them and whatnot. People often tend to roll their own system over it anyway, given not everyone is a fan of using Erlang syntax in configuration files.

{maxT, Milliseconds}

This is the maximum time that the application can run, after which it will be shut down. This is a rather rarely used item and *Milliseconds* defaults to infinity, so you often don't need to bother with this one at all.

{applications, AtomList}

A list of applications on which yours depends. The application system of Erlang will make sure they were loaded and/or started before allowing yours to do so. All applications depend at least on `kernel` and `stdlib`, but if your application were to depend on `ppool` being started, then you should add `ppool` to the list.

Note: yes, the standard library and the VM's kernel are applications themselves, which means that Erlang is a language used to build OTP, but whose runtime environment depends on OTP to work. It's circular. This gives you some idea of why the language is officially named 'Erlang/OTP'.

{mod, {CallbackMod, Args}}

Defines a callback module for the application, using the application behaviour (which we will see in the [next section](#)). This tells OTP that when starting your application, it should call CallbackMod:start(normal, Args). This function's return value will be used when OTP will call CallbackMod:stop(StartReturn) when stopping your application. People will tend to name *CallbackMod* after their application.

And this covers most of what we might need for now (and for most applications you'll ever write).

Converting the Pool

How about we put this into practice? We'll turn the ppool set of processes from last chapter into a basic OTP application. The first step for this is to redistribute everything under the right directory structure. Just create five directories and distribute the files as follows:

```
ebin/  
include/  
priv/  
src/  
  - ppool.erl  
  - ppool_serv.erl
```

```
- ppool_sup.erl  
- ppool_supersup.erl  
- ppool_worker_sup.erl  
test/  
- ppool_tests.erl  
- ppool_nagger.erl
```

You'll notice I moved the `ppool_nagger` to the `test` directory. This is for a good reason — it was not much more than a demo case and would have nothing to do with our application, but is still necessary for the tests. We can actually try it later on once the app has all been packaged so we can make sure everything still works, but for the moment it's kind of useless.

We'll add an `Emakefile` (appropriately named `Emakefile`, placed in the app's base directory) to help us compile and run things later on:

```
{"src/*", [debug_info, {i,"include/"}, {outdir, "ebin/"}]}.  
{"test/*", [debug_info, {i,"include/"}, {outdir, "ebin/"}]}.
```

This just tells the compiler to include `debug_info` for all files in `src/` and `test/`, tells it to go look in the `include/` directory (if it's ever needed) and then shove the files up its `ebin/` directory.

Speaking of which, let's add the `app` file in the `ebin/` directory:

```
{application, ppool,  
[{vsn, "1.0.0"},  
 {modules, [ppool, ppool_serv, ppool_sup, ppool_supersup, ppool_worker_sup]},  
 {registered, [ppool]},  
 {mod, {ppool, []}}  
]}.
```

This one only contains fields we find necessary; env, maxT and applications are not used. We now need to change how the callback module (ppool) works. How do we do that exactly?

First, let's see the application behaviour.

Note: even though all applications depend on the kernel and the stdlib applications, I haven't included them. ppool will still work because starting the Erlang VM starts these applications automatically. You might feel like adding them for the sake of expliciteness, but there's no *need* for it right now.



The Application Behaviour

As for most OTP abstractions we've seen, what we want is a pre-built implementation. Erlang programmers are not happy with design patterns as a convention, they want a solid abstraction for them. This gives us a behaviour for applications. Remember that behaviours are always about splitting generic code away from specific code. They denote the idea that your specific code gives up its own execution flow and inserts itself as a bunch of callbacks to be used by the generic code. In simpler words, behaviours handle the boring parts while you connect the dots.

In the case of applications, this generic part is quite complex and not nearly as simple as other behaviours.

Whenever the VM first starts up, a process called the *application controller* is started (with the name `application_controller`). It starts all other applications and sits on top of most of them. In fact, you could say the application controller acts a bit like a supervisor for all applications. We'll see what kind of supervision strategies there are in the [From Chaos to Application](#) section.

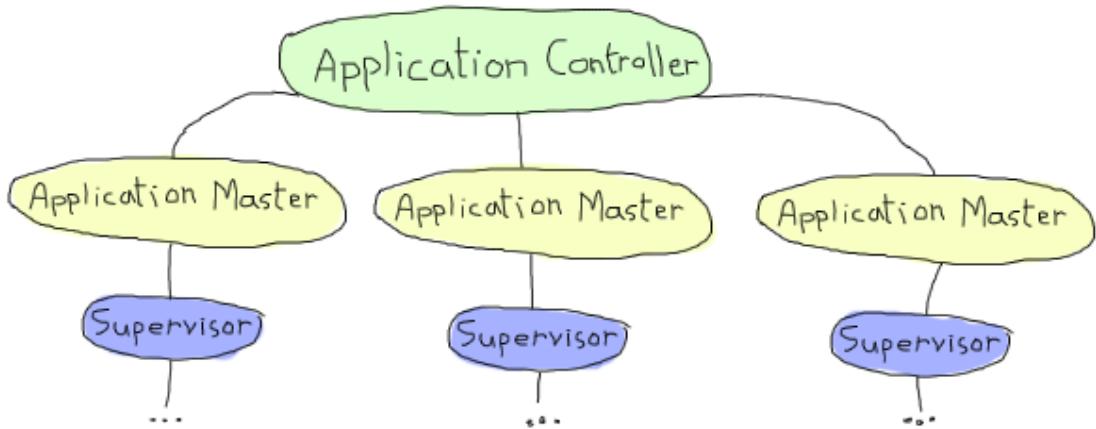
Note: the Application Controller technically doesn't sit over all the applications. One exception is the kernel application, which itself starts a process named `user`. The `user` process in fact acts as a group leader to the application controller and the kernel application thus needs some special treatment. We don't have to care about this, but I felt like it should be included for the sake of precision.

In Erlang, the IO system depends on a concept called a *group leader*. The group leader represents standard input and output and is inherited by all processes. There is a hidden IO protocol that the group leader and any process calling IO functions communicate with. The group leader then takes the responsibility of forwarding these messages to whatever input/output channels there are, weaving some magic that doesn't concern us within the confines of this text.

Anyway, when someone decides they want to start an application, the application controller (often noted AC in OTP parlance) starts an *application master*. The application master is in fact two processes taking charge of each individual

application: they set it up and act like a middleman in between your application's top supervisor and the application controller. OTP is a bureaucracy, and we have many layers of middle-management! I won't get into the details of what happens in there as most Erlang developers will never actually need to care about that and very little documentation exists (the code is the documentation). Just know that the application master acts a bit like the app's nanny (well, a pretty insane nanny). It looks over its children and grandchildren, and when things go awry, it goes berserk and terminates its whole family tree. Brutally killing children is a common topic among Erlangers.

An Erlang VM with a bunch of applications might look a bit like this:



Up to now, we were still looking at the generic part of the behaviour, but what about the specific stuff? After all, this is all we actually have to program. Well the application callback module requires very few functions to be functional: `start/2` and `stop/1`.

The first one takes the form `YourMod:start(Type, Args)`. For now, the `Type` will always be `normal` (the other possibilities accepted have to do with distributed applications, which we'll see at a later point). `Args` is what is coming from your app file. The function initialises everything for your app and only needs to return the `Pid` of the application's top-level supervisor in one of the two following forms: `{ok, Pid}` or `{ok, Pid, SomeState}`. If you don't return `SomeState`, it simply defaults to `[]`.

The `stop/1` function takes the state returned by `start/2` as an argument. It runs after the application is done running and only does the necessary cleanup.

That's it. A huge generic part, a tiny specific one. Be thankful for that, because you wouldn't want to write the rest of things too often (just look at the source if you feel like it!) There are a few more functions that you can optionally use to have more control over the application, but we don't need them for now. This means we can move forward with our ppool application!

From Chaos to Application

We have the app file and a general idea of how applications work. Two simple callbacks. Opening `ppool.erl`, we change the following lines:

```
-export([start_link/0, stop/0, start_pool/3,  
       run/2, sync_queue/2, async_queue/2, stop_pool/1]).
```

```
start_link() ->  
    ppool_supersup:start_link().
```

```
stop() ->  
    ppool_supersup:stop().
```

To the following ones instead:

```
-behaviour(application).  
-export([start/2, stop/1, start_pool/3,  
        run/2, sync_queue/2, async_queue/2, stop_pool/1]).
```

```
start(normal, _Args) ->  
    ppool_supersup:start_link().
```

```
stop(_State) ->  
    ok.
```

We can then make sure the tests are still valid. Pick the old `ppool_tests.erl` file (I wrote it for the previous chapter and am bringing it back here) and replace the single call to `ppool:start_link/0` to `application:start(ppool)` as follows:

```
find_unique_name() ->  
    application:start(ppool),  
    Name = list_to_atom(lists:flatten(io_lib:format("~p",[now()]))),  
    ?assertEqual(undefined, whereis(Name)),  
    Name.
```

You should also take the time to remove `stop/0` from `ppool_supersup` (and remove the export), because the OTP application tools will take care of that for us.

We can finally recompile the code and run all the tests to make sure everything still works (we'll see how that `eunit` thing works later on, don't worry):

```
$ erl -make
Recompile: src/ppool_worker_sup
Recompile: src/ppool_supersup
...
$ erl -pa ebin/
...
1> make:all([load]).
Recompile: src/ppool_worker_sup
Recompile: src/ppool_supersup
Recompile: src/ppool_sup
Recompile: src/ppool_serv
Recompile: src/ppool
Recompile: test/ppool_tests
Recompile: test/ppool_nagger
up_to_date
2> eunit:test(ppool_tests).
All 14 tests passed.
ok
```

The tests take a while to run due to `timer:sleep(x)` being used to synchronise everything in a few places, but it should tell you everything works, as shown above. Good news, our app is healthy.

We can now study the wonders of OTP applications by using our new awesome callbacks:

```
3> application:start(ppool).
ok
4> ppool:start_pool(nag, 2, {ppool_nagger, start_link, []}).
{ok,<0.142.0>}
5> ppool:run(nag, [make_ref(), 500, 10, self()]). 
{ok,<0.146.0>}
6> ppool:run(nag, [make_ref(), 500, 10, self()]). 
{ok,<0.148.0>}
7> ppool:run(nag, [make_ref(), 500, 10, self()]).
```

```
noalloc
9> flush().
Shell got {<0.146.0>,#Ref<0.0.0.625>}
Shell got {<0.148.0>,#Ref<0.0.0.632>}
...
received down msg
received down msg
```

The magic command here is `application:start(ppool)`. This tells the application controller to launch our ppool application. It starts the `ppool_supersup` supervisor and from that point on, everything can be used as normal. We can see all the applications currently running by calling `application:which_applications()`:

```
10> application:which_applications().
[{ppool,[],"1.0.0"},
 {stdlib,"ERTS CXC 138 10","1.17.4"},
 {kernel,"ERTS CXC 138 10","2.14.4"}]
```

What a surprise, ppool is running. As mentioned earlier, we can see that all applications depend on kernel and stdlib, which are both running. If we want to close the pool:

```
11> application:stop(ppool).

=INFO REPORT==== DD-MM-YYYY::23:14:50 ===
  application: ppool
  exited: stopped
  type: temporary
ok
```

And it is done. You should notice that we now get a clean shutdown with a little informative report rather than the messy ** exception exit: killed from last chapter.

Note: You'll sometimes see people do something like `MyApp:start(...)` instead of `application:start(MyApp)`. While this works for testing purposes, it's ruining a lot of the advantages of actually having an application: it's no longer part of the VM's supervision tree, can not access its environment variables, will not check dependencies before being started, etc. Try to stick to `application:start/1` if possible.

Look at this! What's that thing about our app being *temporary*? We write Erlang and OTP stuff because it's supposed to run forever, not just for a while! How dare the VM say this? The secret is that we can give different arguments to `application:start`. Depending on the arguments, the VM will react differently to termination of one of its applications. In some cases, the VM will be a loving beast ready to die for its children. In other cases, it's rather a cold heartless and pragmatic machine willing to tolerate many of its children dying for the survival of its species.

Application started with: `application:startAppName, temporary)`

Ends normally: Nothing special happens, the application has stopped.

Ends abnormally: The error is reported, and the application terminates without restarting.

Application started with: `application:startAppName, transient)`

Ends normally: Nothing special happens, the application has stopped.

Ends abnormally: The error is reported, all the other applications are stopped and the VM shuts down.

Application started with: `application:startAppName, permanent)`

Ends normally: All other applications are terminated and the VM shuts down.

Ends abnormally: Same; all applications are terminated, the VM shuts down.

You can see something new in the supervision strategies when it comes to applications. No longer will the VM try to save you. At this point, something has had to go very, very wrong for it to go up the whole supervision tree of one of its vital applications, enough to crash it. When this does happen, the VM has lost all hope in your program. Given the definition of insanity is to do the same thing all over again while expecting different outcomes each time, the VM prefers to die sanely and just give up. Of course the real reason has to do with something being broken that needs to be fixed, but you catch my drift. Take note that all applications can be terminated by calling `application:stop(AppName)` without affecting others as if a crash had occurred.

Library Applications

What happens when we want to wrap flat modules in an application but we have no process to start and thus no need for an application callback module?

After pulling our hair and crying in rage for a few minutes, the only other thing left to do is to remove the tuple `{mod, {Module, Args}}` from the application file. That's it. This is called a *library application*. If you want an example of one, the Erlang stdlib (standard library) application is one of these.

If you have the source package of Erlang, you can go to `otp_src_<release>/lib/stdlib/src/stdlib.app.src` and see the following:

```
{application, stdlib,  
 [{description, "ERTS CXC 138 10"},
```

```
{vsn, "%VSN%"},  
{modules, [array,  
          ...  
          gen_event,  
          gen_fsm,  
          gen_server,  
          io,  
          ...  
          lists,  
          ...  
          zip]},  
{registered,[timer_server,rsh_starter,take_over_monitor,pool_master,  
           dets]},  
{applications, [kernel]},  
{env, []]}].
```

You can see it's a pretty standard application file, but without the callback module. A library application.

How about we go deeper with applications?

The Count of Applications

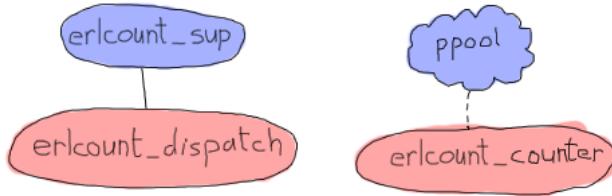
From OTP Application to Real Application



Our ppool app has become a valid OTP application and we are now able to understand what this means. It would be nice to build something that actually uses our process pool to do anything useful, though. To push our knowledge of applications a bit further, we will write a second application. This one will depend on ppool, but will be able to benefit from some more automation than our 'hagger'.

This application, that I will name `erlcount`, will have a somewhat simple objective: recursively look into some directory, find all Erlang files (ending in `.erl`) and then run a regular expression over it to count all instances of a given string within the modules. The results are then accumulated to give the final result, which will be output to the screen.

This specific application will be relatively simple, relying heavily on our process pool instead. It will have a structure as follows:



In the diagram above, ppool represents the whole application, but only means to show that `erlcount_counter` will be the worker for the process pool. This one will open files, run the regular expression and return the count. The process/module `erlcount_sup` will be our supervisor, and `erlcount_dispatch` will be a single server in charge of browsing the directories, asking ppool to schedule workers and compiling the results. We'll also add an `erlcount_lib` module, taking charge of hosting all the functions to read directories, compile data and whatnot, leaving the other modules with the responsibility of coordinating these calls. Last will be an `erlcount` module with the single purpose of being the application callback module.

The first step, as for our last app, is to create the directory structure needed. You can also add a few file stubs if you feel like doing so:

```
ebin/  
  - erlcount.app  
include/  
priv/  
src/  
  - erlcount.erl  
  - erlcount_counter.erl  
  - erlcount_dispatch.erl  
  - erlcount_lib.erl  
  - erlcount_sup.erl  
test/  
Emakefile
```

Nothing too different from what we had before, you can even copy the Emakefile we had before.

We can probably start writing most parts of the application pretty quickly. The .app file, the counter, library and supervisor should be relatively simple. On the other hand, the dispatch module will have to accomplish some complex tasks if we want things to be worth it. Let's start with the app file:

```
{application, erlcount,  
 [{vsn, "1.0.0"},  
 {modules, [erlcount, erlcount_sup, erlcount_lib,  
          erlcount_dispatch, erlcount_counter]}],  
 {applications, [ppool]},  
 {registered, [erlcount]},  
 {mod, {erlcount, []}},  
 {env,  
  [{directory, ":"},  
   {regex, ["if\\ \\s.+->", "case\\ \\s.+\\ \\sof"]},  
   {max_files, 10}]}  
].
```

This app file is a bit more complex than the ppool one. We can still recognize some of the fields as being the same: this app will also be in version 1.0.0, the modules listed are all the same as above. The next part is something we didn't have: an application dependency. As explained earlier, the applications tuple gives a list of all the applications that should be started before erlcount. If you try to start it without that, you'll get an error message. We then have to count the registered processes with {registered, [erlcount]}. Technically none of our modules started as part of the erlcount app will need a name. Everything we do can be anonymous. However, because we know ppool registers the ppool_serv to the name we give it and because we know we will use a process pool, then we're going to call it erlcount and note it there. If all applications that use ppool do the same, we should be able to detect conflicts in

the future. The `mod` tuple is similar as before; we define the application behaviour callback module there.



The last new thing in here is the `env` tuple. As seen earlier, this entire tuple gives us a key/value store for application-specific configuration variables. These variables will be accessible from all the processes running within the application, stored in memory for your convenience. They can basically be used as substitute configuration file for your app.

In this case, we define three variables: `directory`, which tells the app where to look for `.erl` files (assuming we run the app from the `erlcunt-1.0` directory, this means the `learn-you-some-erlang`), then we have `max_files` telling us how many file descriptors should be opened at once. We don't want to open 10,000 files at once if we end up have that many, so this variable will match the maximum number of workers in `ppool`. Then the most complex variable is `regex`. This one will contain a list of all regular expressions we want to run over each of the files to count the results.

I won't get into the long explaining of the syntax of Perl Compatible Regular Expressions (if you're interested, the `re` module contains some documentation), but will still explain what we're doing here. In this case, the first regular expression says "look for a string that contains 'if' followed by any single white space character (`\s`, with a second backslash for escaping purposes), and finishes with `->`. Moreover there can be anything in between the 'if' and the `->` `(.+)`". The second regular expression says "look for a string that contains 'case' followed by any single whitespace character (`\s`), and finishes with 'of' preceded by single whitespace character. Between the 'case' and the 'of', there can be anything `(.+)`". To make things simple, we'll try to count how many times we use `case ... of` vs. how many times we use `if ... end` in our libraries.

Don't Drink Too Much Kool-Aid:

Using regular expressions is not an optimal choice to analyse Erlang code. The problem is there are lots of cases that will make our results inaccurate, including strings in the text and comments that match the patterns we're looking for but are technically not code.

To get more accurate results, we would need to look at the parsed and the expanded version of our modules, directly in Erlang. While more complex (and outside the scope of

this text), this would make sure that we handle everything like macros, exclude comments, and just generally do it the right way.

With this file out of the way, we can start the application callback module. It won't be complex, basically only starting the supervisor:

```
-module(erlcount).
-behaviour(application).
-export([start/2, stop/1]).

start(normal, _Args) ->
    erlcount_sup:start_link().

stop(_State) ->
    ok.
```

And now the supervisor itself:

```
-module(erlcount_sup).
-behaviour(supervisor).
-export([start_link/0, init/1]).

start_link() ->
    supervisor:start_link(?MODULE, []).

init([]) ->
    {ok, {{one_for_one, MaxRestart, MaxTime},
          [{dispatch,
            {erlcount_dispatch, start_link, []},
            transient,
            60000,
            worker,
            [erlcount_dispatch]}}]}.
```

This is a standard supervisor, which will be in charge of only `erlcount_dispatch`, as it was shown on the previous little schema. The `MaxRestart`, `MaxTime` and the 60 seconds value for shutdown were chosen pretty randomly, but in real cases you'd want to study the needs you have. Because this is a demo application, it didn't seem that important at the time. The author keeps himself the right to laziness.

We can get to the next process and module in the chain, the dispatcher. The dispatcher will have a few complex requirements to fulfill for it to be useful:

- When we go through directories to find files ending in `.erl`, we should only go through the whole list of directories once, even when we apply multiple regular expressions;
- We should be able to start scheduling files for result counting as soon as we find there's one that matches our criteria. We should not need to wait for a complete list to do so.

- We need to hold a counter per regular expression so we can compare the results in the end
- It is possible we start getting results from the `erlcount_counter` workers before we're done looking for `.erl` files
- It is possible that many `erlcount_counters` will be running at once
- It is likely we will keep getting result after we finished looking files up in the directories (especially if we have many files or complex regular expressions).

The two big points we have to consider right now is how we're going to go through a directory recursively while still being able to get results from there in order to schedule them, and then accept results back while that goes on, without getting confused.



At a first look, the way that looks the simplest to gain the ability to return results while in the middle of recursion would be to use a process to do it. However, it's a bit annoying to change our previous structure just to be able to add another process to the supervision tree, then to get them working together. There is, in fact, a simpler way to do things.

This is a style of programming called *Continuation-Passing Style*. The basic idea behind it is to take one function that's usually deeply recursive and break every step down. We return each step (which would usually be the accumulator), and then a function that will allow us to keep going after that. In our case, our function will basically have two possible return values:

```
{continue, Name, NextFun}
done
```

Whenever we receive the first one, we can schedule `FileName` into `ppool` and then call `NextFun` to keep looking for more files. We can implement this function into `erlcount_lib`:

```
-module(erlcount_lib).
-export([find_erl/1]).
-include_lib("kernel/include/file.hrl").

%% Finds all files ending in .erl
find_erl(Directory) ->
    find_erl(Directory, queue:new()).
```

Ah, something new there! What a surprise, my heart is racing and my blood is pumping. The include file up there is something given to us by the `file` module. It contains a record (`#file_info{}`) with a bunch of fields explaining details about the file, including its type, size, permissions, and so on.

Our design here includes a queue. Why is that? Well it is entirely possible that a directory contains more than one file. So when we hit a directory and it contains something like 15 files, we want to handle the first one (and if it's a directory, open it, look inside, etc.) and then handle the 14 others later. In order to do so, we will just store their names in memory until we have the time process them. We use a queue for that, but a stack or any other data structure would still be fine given we don't really care about the order in which we read files. Anyway, the point is, this queue acts a bit like a to-do list for files in our algorithm.

Alright so let's start by reading the first file passed from the first call:

```
%%% Private
%% Dispatches based on file type
find_erl(Name, Queue) ->
    {ok, F = #file_info{}} = file:read_file_info(Name),
    case F#file_info.type of
        directory -> handle_directory(Name, Queue);
        regular -> handle_regular_file(Name, Queue);
        _Other -> dequeue_and_run(Queue)
    end.
```

This function tells us few things: we only want to deal with regular files and directories. In each case we will write ourselves a function to handle these specific occurrences (`handle_directory/2` and `handle_regular_file/2`). For other files, we will dequeue anything we had prepared before with the help of `dequeue_and_run/1` (we'll see what this one is about soon). For now, we first start dealing with directories:

```
%% Opens directories and enqueues files in there
handle_directory(Dir, Queue) ->
    case file:list_dir(Dir) of
        {ok, []} ->
            dequeue_and_run(Queue);
        {ok, Files} ->
            dequeue_and_run(enqueue_many(Dir, Files, Queue))
    end.
```

So if there are no files, we keep searching with `dequeue_and_run/1`, and if there are many, we enqueue them before doing so. Let me explain this. The function `dequeue_and_run` will take the queue of file names and get one element out of it. The file name it fetches out from there will be used by calling `find_erl(Name, Queue)` and we just keep going as if we were just getting started:

```
%% Pops an item from the queue and runs it.
dequeue_and_run(Queue) ->
```

```

case queue:out(Queue) of
    {empty, _} -> done;
    {{value, File}, NewQueue} -> find_erl(File, NewQueue)
end.

```

Note that if the queue is empty ({empty, _}), the function considers itself done (a keyword chosen for our CPS function), otherwise we keep going over again.

The other function we had to consider was `enqueue_many/3`. This one is designed to enqueue all the files found in a given directory and works as follows:

```

%% Adds a bunch of items to the queue.
enqueue_many(Path, Files, Queue) ->
    F = fun(File, Q) -> queue:in(filename:join(Path, File), Q) end,
    lists:foldl(F, Queue, Files).

```

Basically, we use the function `filename:join/2` to merge the directory's path to each file name (so that we get a complete path). We then add this new full path to a file to the queue. We use a fold to repeat the same procedure with all the files in a given directory. The new queue we get out of it is then used to run `find_erl/2` again, but this time with all the new files we found added to the to-do list.

Whoa, we digressed a bit. Where were we? Oh yes, we were handling directories and now we're done with them. We then need to check for regular files and whether they end in `.erl` or not.

```

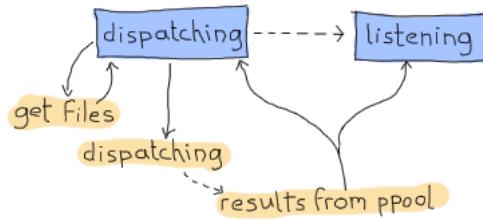
%% Checks if the file finishes in .erl
handle_regular_file(Name, Queue) ->
    case filename:extension(Name) of
        ".erl" ->
            {continue, Name, fun() -> dequeue_and_run(Queue) end};
        _NonErl ->
            dequeue_and_run(Queue)
    end.

```

You can see that if the name matches (according to `filename:extension/1`), we return our continuation. The continuation gives the `Name` to the caller, and then wraps the operation `dequeue_and_run/1` with the queue of files left to visit into a fun. That way, the user can call that fun and keep going as if we were still in the recursive call, while still getting results in the mean time. In the case where the file name doesn't end in `.erl`, then the user has no interest in us returning yet and we keep going by dequeuing more files. That's it.

Hooray, the CPS thing is done. We can then focus on the other issue. How are we going to design the dispatcher so that it can both dispatch and receive at once? My suggestion, which you will no doubt accept because I'm the one writing the text, is to use a finite state machine. It will have two states. The first one will be the 'dispatching' state. It's the one used whenever we're waiting for our `find_erl` CPS function to hit the `done` entry. While we're in there,

we will never think about us being done with the counting. That will only happen in the second and final state, 'listening', but we will still receive notices from ppool all the time:



This will thus require us to have:

1. A dispatching state with an asynchronous event for when we get new files to dispatch
2. A dispatching state with an asynchronous event for when we are done getting new files
3. A listening state with an asynchronous event for when we're done getting new files
4. A global event to be sent by the ppool workers when they're done running their regular expression.

We'll slowly start building our gen_fsm:

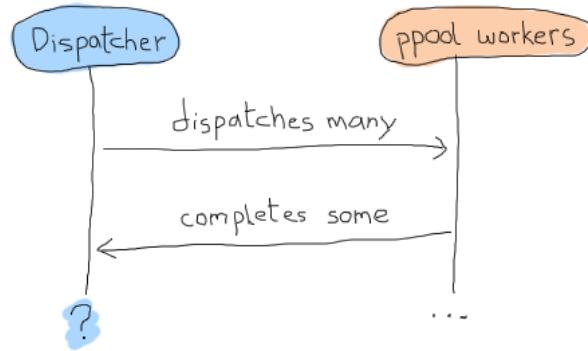
```

-module(erlcount_dispatch).
-behaviour(gen_fsm).
-export([start_link/0, complete/4]).
-export([init/1, dispatching/2, listening/2, handle_event/3,
        handle_sync_event/4, handle_info/3, terminate/3, code_change/4]).

-define(Pool, erlcount).
  
```

Our API will thus have two functions: one for the supervisor (`start_link/0`) and one for the ppool callers (`complete/4`, we'll see the arguments when we get there). The other functions are the standard gen_fsm callbacks, including our `listening/2` and `dispatching/2` asynchronous state handlers. I also defined a `?POOL` macro, used to give our ppool server the name 'erlcount'.

What should the gen_fsm's data look like, though? Because we're going asynchronous and we are going to always call `ppool:run_async/2` instead of anything else, we will have no real way of knowing if we're ever done scheduling files or not. Basically we could have a timeline like this:



One way to solve the problem could be to use a timeout, but this is always annoying: is the timeout too long or too short? Has something crashed? This much uncertainty is probably as fun as a toothbrush made of lemon. Instead, we could use a concept where each worker is given some kind of identity, which we can track and associate with a reply, a bit like a secret password to enter the private club of 'workers who succeeded'. This concept will let us match one-on-one whatever message we get and let us know when we are absolutely done. We now know what our state data might look like this:

```
-record(data, {regex=[], refs=[]}).
```

The first list will be tuples of the form `{RegularExpression, NumberOfOccurrences}`, while the second will be a list of some kind of references to the messages. Anything will do, as long as it's unique. We can then add the two following API functions:

```
%%% PUBLIC API
start_link() ->
    gen_fsm:start_link(?MODULE, [], []).

complete(Pid, Regex, Ref, Count) ->
    gen_fsm:send_all_state_event(Pid, {complete, Regex, Ref, Count}).
```

And here is our secret `complete/4` function. Unsurprisingly, the workers will only have to send back 3 pieces of data: what regular expression they were running, what their associated score was, and then the reference mentioned above. Awesome, we can get into the real interesting stuff!

```
init([]) ->
    %% Move the get_env stuff to the supervisor's init.
    {ok, Re} = application:get_env(regex),
    {ok, Dir} = application:get_env(directory),
    {ok, MaxFiles} = application:get_env(max_files),
    ppool:start_pool(?POOL, MaxFiles, {erlcount_counter, start_link, []}),
    case lists:all(fun valid_regex/1, Re) of
        true ->
            self() ! {start, Dir},
            {ok, dispatching, #data{regex=[{R,0} || R <- Re]}};
        false ->
```

```

    {stop, invalid_regex}
end.

```

The init function first loads all the info we need to run from the application file. Once that's done, we plan on starting the process pool with erlcount_counter as a callback module. The last step before actually going is to make sure all regular expressions are valid. The reason for this is simple. If we do not check it right now, then we will have to add error handling call somewhere else instead. This is likely going to be in the erlcount_counter worker. Now if it happens there, we now have to define what do we do when the workers start crashing because of that and whatnot. It's just simpler to handle when starting the app. Here's the valid_regex/1 function:

```

valid_regex(Re) ->
    try re:run("", Re) of
        _ -> true
    catch
        error:badarg -> false
    end.

```

We only try to run the regular expression on an empty string. This will take no time and let the `re` module try and run things. So the regexes are valid and we start the app by sending ourselves `{start, Directory}` and with a state defined by `[{R,0} || R <- Re]`. This will basically change a list of the form `[a,b,c]` to the form `[{a,0},{b,0},{c,0}]`, the idea being to add a counter to each of the regular expressions.

The first message we have to handle is `{start, Dir}` in `handle_info/2`. Remember, because Erlang's behaviours are pretty much all based on messages, we have to do the ugly step of sending ourselves messages if we want to trigger a function call and do things our way. Annoying, but manageable:

```

handle_info({start, Dir}, State, Data) ->
    gen_fsm:send_event(self(), erlcount_lib:find_erl(Dir)),
    {next_state, State, Data}.

```

We send ourselves the result of `erlcount_lib:find_erl(Dir)`. It will be received in the dispatching, given that's the value of `State`, as it was set by the `init` function of the FSM. This snippet solves our problem, but also illustrates the general pattern we'll have during the whole FSM. Because our `find_erl/1` function is written in a Continuation-Passing Style, we can just send ourselves an asynchronous event and deal with it in each of the right callback states. It is likely that the first result of our continuation will be `{continue, File, Fun}`. We will also be in the 'dispatching' state, because that's what we put as the initial state in the `init` function:

```

dispatching({continue, File, Continuation}, Data = #data{regex=Re, refs=Refs}) ->
    F = fun({Regex, _Count}, NewRefs) ->
        Ref = make_ref(),
        ppool:async_queue(?POOL, [self(), Ref, File, Regex]),
        [Ref|NewRefs]
    end,

```

```

NewRefs = lists:foldl(F, Refs, Re),
gen_fsm:send_event(self(), Continuation()),
{next_state, dispatching, Data#data{refs = NewRefs}};

```

That's a bit ugly. For each of the regular expressions, we create a unique reference, schedule a ppool worker that knows this reference, and then store this reference (to know if a worker has finished). I chose to do this in a foldl in order to make it easier to accumulate all the new references. Once that dispatching is done, we call the continuation again to get more results, and then wait for the next message with the new references as our state.

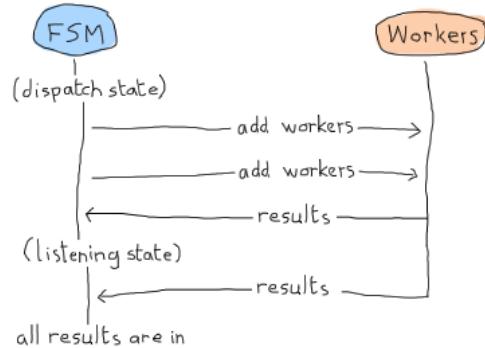
What's the next kind of message we can get? We have two choices here. Either none of the workers have given us our results back (even though they have not been implemented yet) or we get the done message because all files have been looked up. Let's go the second type to finish implementing the dispatching/2 function:

```

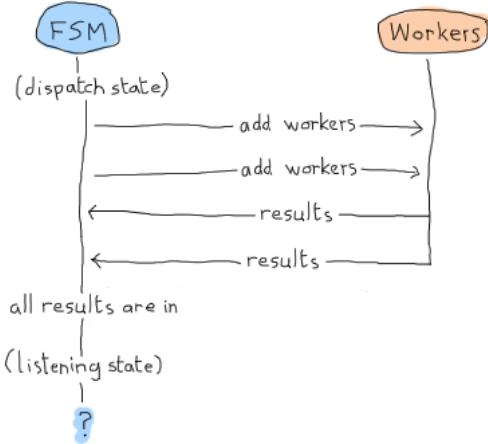
dispatching(done, Data) ->
%% This is a special case. We can not assume that all messages have NOT
%% been received by the time we hit 'done'. As such, we directly move to
%% listening/2 without waiting for an external event.
listening(done, Data).

```

The comment is pretty explicit as to what is going on, but let me explain anyway. When we schedule jobs, we can receive results while in dispatching/2 or while in listening/2. This can take the following form:



In this case, the 'listening' state can just wait for results and declare everything is in. But remember, this is Erlang Land (*Erland*) and we work in parallel and asynchronously! This scenario is as probable:



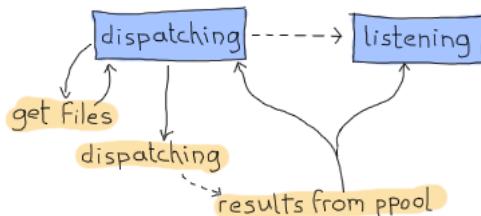
Ouch. Our application would then be hanging forever, waiting for messages. This is the reason why we need to manually call `listening/2`: we will force it to do some kind of result detection to make sure everything has been received, just in case we already have all the results. Here's what this looks like:

```

listening(done, #data{regex=Re, refs=[]}) -> % all received!
  [io:format("Regex ~s has ~p results~n", [R,C]) || {R, C} <- Re],
  {stop, normal, done};
listening(done, Data) -> % entries still missing
  {next_state, listening, Data}.

```

If no `refs` are left, then everything was received and we can output the results. Otherwise, we can keep listening to messages. If you take another look at `complete/4` and this diagram:



The result messages are global, because they can be received in either 'dispatching' or 'listening' states. Here's the implementation:

```

handle_event({complete, Regex, Ref, Count}, State, Data = #data{regex=Re, refs=Refs}) ->
  {Regex, OldCount} = lists:keyfind(Regex, 1, Re),
  NewRe = lists:keyreplace(Regex, 1, Re, {Regex, OldCount+Count}),
  NewData = Data#data{regex=NewRe, refs=Refs--[Ref]},
  case State of
    dispatching ->
      {next_state, dispatching, NewData};
    listening ->
      listening(done, NewData)
  end.

```

The first thing this does is find the regular expression that just completed in the *Re* list, which also contains the count for all of them. We extract that value (*OldCount*) and update it with the new count (*oldCount+Count*) with the help of `lists:keyreplace/4`. We update our *Data* record with the new scores while removing the *Ref* of the worker, and then send ourselves to the next state.

In normal FSMs, we would just have done `{next_state, State, NewData}`, but here, because of the problem mentioned with regards to knowing when we're done or not, we have to manually call `listening/2` again. Such a pain, but alas, a necessary step.

And that's it for the dispatcher. We just add in the rest of the filler behaviour functions:

```
handle_sync_event(Event, _From, State, Data) ->
    io:format("Unexpected event: ~p~n", [Event]),
    {next_state, State, Data}.

terminate(_Reason, _State, _Data) ->
    ok.

code_change(_OldVsn, State, Data, _Extra) ->
    {ok, State, Data}.
```

And we can then move on to the counter. You might want to take a little break before then. Hardcore readers can go bench press their own weight a few times to relax themselves and then come back for more.

The Counter

The counter is simpler than the dispatcher. While we still need a behaviour to do things (in this case, a `gen_server`), it will be quite minimalist. We only need it to do three things:

1. Open a file
2. Run a regex on it and count the instances
3. Give the result back.

For the first point, we have plenty of functions in `file` to help us do that. For the number 3, we defined `erlcount_dispatch:complete/4` to do it. For the number 2, we can use the `re` module with `run/2-3`, but it doesn't quite do what we need:

```
1> re:run(<<"brutally kill your children (in Erlang)">>, "a").
{match,[{4,1}]}
2> re:run(<<"brutally kill your children (in Erlang)">>, "a", [global]).
{match,[[{4,1}],[{35,1}]]}
3> re:run(<<"brutally kill your children (in Erlang)">>, "a", [global, {capture, all, list}]). 
{match,[{"a","a"}]}
4> re:run(<<"brutally kill your children (in Erlang)">>, "child", [global, {capture, all, list}]). 
{match,[{"child"}]}
```

While it does take the arguments we need (`re:run(String, Pattern, Options)`), it doesn't give us the correct count. Let's add the following function to `erlcount_lib` so we can start writing the counter:

```
regex_count(Re, Str) ->
    case re:run(Str, Re, [global]) of
        nomatch -> 0;
        {match, List} -> length(List)
    end.
```

This one basically just counts the results and returns that. Don't forget to add it to the export form.

Ok, on with the worker:

```
-module(erlcount_counter).
-behaviour(gen_server).
-export([start_link/4]).
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
        terminate/2, code_change/3]).  
  
-record(state, {dispatcher, ref, file, re}).  
  
start_link(DispatcherPid, Ref, FileName, Regex) ->
    gen_server:start_link(?MODULE, [DispatcherPid, Ref, FileName, Regex], []).  
  
init([DispatcherPid, Ref, FileName, Regex]) ->
    self() ! start,  
    {ok, #state{dispatcher=DispatcherPid,  
               ref = Ref,  
               file = FileName,  
               re = Regex}}.  
  
handle_call(_Msg, _From, State) ->
    {noreply, State}.  
  
handle_cast(_Msg, State) ->
    {noreply, State}.  
  
handle_info(start, S = #state{re=Re, ref=Ref}) ->
    {ok, Bin} = file:read_file(S#state.file),
    Count = erlcount_lib:regex_count(Re, Bin),
    erlcount_dispatch:complete(S#state.dispatcher, Re, Ref, Count),
    {stop, normal, S}.  
  
terminate(_Reason, _State) ->
    ok.  
  
code_change(_OldVsn, State, _Extra) ->
    {ok, State}.
```

The two interesting sections here are the `init/1` callback, where we order ourselves to start, and then a single `handle_info/2` clause where we open the file (`file:read_file(Name)`), get a binary back, which we pass to our new `regex_count/2` function, and then send it back with `complete/4`. We then stop the worker. The rest is just standard OTP callback stuff.

We can now compile and run the whole thing!

```
$ erl -make  
Recompile: src/erlcount_sup  
Recompile: src/erlcount_lib  
Recompile: src/erlcount_dispatch  
Recompile: src/erlcount_counter  
Recompile: src/erlcount  
Recompile: test/erlcount_tests
```

Hell yes. Pop the champagne because we have no whine!

Run App Run

There are many ways to get our app running. Make sure you're in a directory where you somehow have these two directories next to each other:

```
erlcount-1.0  
ppool-1.0
```

Now start Erlang the following way:

```
$ erl -env ERL_LIBS ":"
```

The `ERL_LIBS` variable is a special variable defined in your environment that lets you specify where Erlang can find OTP applications. The VM is then able to automatically look in there to find the `ebin/` directories for you. `erl` can also take an argument of the form `-env NameOFVar Value` to override this setting quickly, so that's what I used here. The `ERL_LIBS` variable is pretty useful, especially when installing libraries, so try to remember it!

With the VM we started, we can test that the modules are all there:

```
1> application:load(ppool).  
ok
```

This function will try to load all the application modules in memory if they can be found. If you don't call it, it will be done automatically when starting the application, but this provides an easy way to test our paths. We can start the apps:

```
2> application:start(ppool), application:start(erlcount).  
ok  
Regex if\s\+\-\> has 20 results  
Regex case\s\+\|sof has 26 results
```

Your results may vary depending on what you have in your directories. Note that depending how many files you have, this can take longer.



What if we want different variables to be set for our applications, though? Do we need to change the application file all the time? No we don't! Erlang also supports that. So let's say I wanted to see how many times the Erlang programmers are angry in their source files?

The `erl` executable supports a special set of arguments of the form `-AppName Key1 Val1 Key2 Val2 ... KeyN ValN`. In this case, we could then run the following regular expression over the Erlang source code from the R14B02 distribution with 2 regular expressions as follows:

```
$ erl -env ERL_LIBS "." -erlcount directory "/home/ferd/otp_src_R14B02/lib/" regex '["shit","damn"]'  
...  
1> application:start(ppool), application:start(erlcount).  
ok  
Regex shit has 3 results  
Regex damn has 1 results  
2> q().  
ok
```

Note that in this case, all expressions I give as arguments are wrapped in single quotation marks (''). That's because I want them to be taken literally by my Unix shell. Different shells might have different rules.

We could also try our search with more general expressions (allowing values to start with capital letters) and with more file descriptors allowed:

```
$ erl -env ERL_LIBS "." -erlcount directory "/home/ferd/otp_src_R14B02/lib/" regex '["[Ss]hit","[Dd]amn"]' max_files 50  
...  
1> application:start(ppool), application:start(erlcount).  
ok  
Regex [Ss]hit has 13 results  
Regex [Dd]amn has 6 results  
2> q().  
ok
```

Oh, OTP programmers. What makes you so angry? ("Working with Erlang" not being an acceptable answer)

This one might take even longer to run due to the more complex checks required over the hundreds of files there. This all works pretty good, but there are a few annoying things there.

Why are we always manually starting both applications? isn't there something better?

Included Applications

Included applications are one way to get things working. The basic idea of an included application is that you define an application (in this case ppool) as an application that is part of another one (erlcount, here). To do this, a bunch of changes need to be made to both applications.

The gist of it is that you modify your application file a bit, and then you need to add something called *start phases* to them, etc.



It is more and more recommended **not** to use included applications for a simple reason: they seriously limit code reuse. Think of it this way. We've spent a lot of time working on ppool's architecture to make it so anybody can use it, get their own pool and be free to do whatever they want with it. If we were to push it into an included application, then it can no longer be included in any other application on this VM, and if erlcount dies, then ppool will be taken down with it, ruining the work of any third party application that wanted to use ppool.

For these reasons, included applications are usually excluded from many Erlang programmers' toolbox. As we will see in the following chapter, releases can basically help us do the same (and much more) in a more generic manner.

Before that, we have a one more topic left to discuss in applications though.

Complex Terminations

There are cases where we need more steps to be done before terminating our application. The `stop/1` function from the application callback module might not be enough, especially since it gets called **after** the application has already terminated. What do we do if we need to clean things up before the application is actually gone?

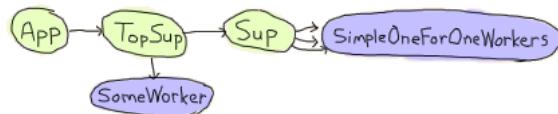
The trick is simple. Just add a function `prep_stop(State)` to your application callback module. `State` will be the state returned by your `start/2` function, and whatever `prep_stop/1` returns will be passed to `stop/1`. The function `prep_stop/1` thus technically inserts itself between `start/2` and `stop/1` and is executed while your application is still alive, but just before it shuts down.

This is the kind of callback that you will know when you need to use it, but that we don't require for our application right now.

Don't drink too much Kool-Aid:

A real world use case of the `prep_stop/1` callback came to me when I was helping Yurii Rashkosvkii (yrashk) debug a problem with agner, a package manager for Erlang. The problems encountered are a bit complex and have to do with weird interactions between simple_one_for_one supervisors and the application master, so feel free to skip this part of the text.

Agner is basically structured in a way where the application is started, starts a top-level supervisor, which starts a server and another supervisor, which in turn spawns the dynamic children



Now the thing is that the documentation says the following:

Important note on simple-one-for-one supervisors: The dynamically created child processes of a simple-one-for-one supervisor are not explicitly killed, regardless of shutdown strategy, but are expected to terminate when the supervisor does (that is, when an exit signal from the parent process is received).

And indeed they are not. The supervisor just kills its regular children and then disappears, leaving it to the simple-one-for-one children's behaviours to catch the exit message and leave. This, alone is fine.

As seen earlier, for each application, we have an application master. This application master acts as a group leader. As a reminder, the application master is linked both to its parent (the application controller) and its direct child (the app's top-level supervisor) and monitors both of them. When any of them fails, the master terminates its own execution, using its status as a group leader to terminate all of the leftover children. Again, this alone is fine.

However, if you mix in both features, and then decide to shut the application down with `application:stop(agner)`, you end up in a very troublesome situation:



At this precise point in time, both supervisors are dead, as well as the regular worker in the app. The simple-one-for-one workers are currently dying, each catching the `EXIT` signal sent

by their direct ancestor.

At the same time, though, The application master gets wind of its direct child dying and ends up killing every one of the simple-one-for-one workers that weren't dead yet.

The result is a bunch of workers which managed to clean up after themselves, and a bunch of others that didn't manage to do so. This is highly timing dependent, hard to debug and easy to fix.

Yurii and I basically fixed this by using the `ApplicationCallback:prep_stop(State)` function to fetch a list of all the dynamic simple-one-for-one children, monitor them, and then wait for all of them to die in the `stop(State)` callback function. This forces the application controller to stay alive until all of the dynamic children were dead. You can see the actual file on Agner's github repository



What an ugly thing! Hopefully, people very rarely run into this kind of issue and you hopefully won't. You can go put some soap in your eyes to wash away the terrible pictures of using `prep_stop/1` to get things working, even though it sometimes makes sense and is desirable. When you're back, we're going to start thinking about packaging our applications into releases.

update:

Since version R15B, the issue above has been resolved. The termination of dynamic children appears to be synchronous in the case of a supervisor shutdown.

Release is the Word

Am I an Executable Yet?

How far have we got. All this work, all these concepts, and we haven't shipped a single Erlang executable yet. You might agree with me that getting an Erlang system up and running requires a lot of effort, especially compared to many languages where you call the compiler and off you go.



Of course this is entirely right. We can compile files, run applications, check for some dependencies, handle crashes and whatnot, but it's not very useful without a functioning Erlang system you can easily deploy or ship with it. What use is it to have great pizza when it can only be delivered cold? (people who enjoy cold pizza might feel excluded here. I am sorry.)

The OTP team didn't leave us on our own when it comes to making sure real systems come to life. OTP releases are part of a system made to help package applications with the minimal resources and dependencies.

Fixing The Leaky Pipes

For our first release, we will reuse our `ppool` and `erlcount` applications from last chapters. However, before we do so, we'll need to change a few things here and there. If you're following along with the book and writing your own code, you might want to copy both of our apps into a new directory called `release/`, which I will assume you will have done for the rest of the chapter.



The first thing that's really bothering me with `erlcount` is that once it's done running, the VM stays up, doing nothing. We might want most applications to stay running forever, but this time it's not the case. Keeping it running made sense because we might have wanted to

play with a few things in the shell and needed to manually start applications, but this should no longer be necessary.

For this reason, we'll add a command that will shut the BEAM virtual machine down in an orderly manner. The best place to do it is within erlcoun_dispatch.erl's own terminate function, given it's called after we obtain the results. The perfect function to tear everything down is init:stop/0. This function is quite complex, but will take care of terminating our applications in order, will get rid of file descriptors, sockets, etc. for us. The new stop function should now look like this:

```
terminate(_Reason, _State, _Data) ->  
    init:stop().
```

And that's it for the code itself. We've got a bit more work to do, still. When we defined our app files during the two last chapters, we did so while using the absolute minimal amount of information necessary to get them running. A few more fields are required so that Erlang isn't completely mad at us.

First of all, the Erlang tools to build releases require us to be a little bit more precise in our application descriptions. You see, although tools for releases don't understand documentation, they still have this intuitive fear of code where the developers were too impolite to at least leave an idea of what the application does. For this reason, we'll need to add a description tuple to both our ppool.app and erlcoun.app files.

For ppool, add the following one:

```
{description, "Run and enqueue different concurrent tasks"}
```

and for erlcoun:

```
{description, "Run regular expressions on Erlang source files"}
```

Now we'll be able to get a better idea of what's going on when we inspect our different systems.

The most attentive readers will also remember I've mentioned at some point that *all* applications depend on stdlib and kernel. However, our two app files do not mention any of these. Let's add both applications to each of our app files. This will require to add the following tuple to the ppool:

```
{applications, [stdlib, kernel]}
```

And add the two applications to the existing erlcoun app file, giving us {applications, [stdlib, kernel, ppool]}.

Don't Drink Too Much Kool-Aid:

While this might have virtually no impact when we start releases manually (and even when

we generate them with systools, which we'll see very soon), it is absolutely vital to add both libraries to the list.

People who generate releases with `reltool` (another tool we'll see in this chapter) will definitely need these applications in order for their release to run well, and even to be able to shut the VM down in a respectable manner. I'm not kidding, it's this necessary. I forgot to do it when writing this chapter and lost a night of work trying to find what the hell was wrong when it was just me not doing things right in the first place.

It could be argued that ideally, the release systems of Erlang could implicitly add these applications given pretty much all of them (except very special cases) will depend on them. Alas, they don't. We'll have to make do with this.

We've got a termination in place and we have updated the app files and whatnot. The last step before we start working with releases is to *compile all your applications*. Successively run your Emakefiles (with `erl -make`) in each directory containing one. Otherwise, Erlang's tools won't do it for you and you'll end up with a release without code to run. Ouch.

Releases With Systools

The systools application is the simplest one to build Erlang releases. It's the *Easy-Bake Oven*® of Erlang releases. To get your delicious releases out of the systools oven, you first need a basic recipe and list of ingredients. If I were to manually describe the ingredients of a successful minimal Erlang release for our `erlcount` application, it'd look a bit like this:

Ingredients for `erlcount` 1.0.0

- An Erlang Run Time System (ERTS) of your choice.
- A standard library
- A kernel library
- The `ppool` application, which should not fail
- The `erlcount` application.

Did I mention that I'm a terrible cook? I'm not sure I can even cook pancakes, but at least I know how to build an OTP release. The ingredient list for an OTP release with `systools` looks like this file, named `erlcount-1.0.rel` and placed at the top-level of the `release/` directory:

```
{release,  
 {"erlcount", "1.0.0"},  
 {erts, "5.8.4"},  
 [{kernel, "2.14.4"},  
 {stdlib, "1.17.4"},  
 {ppool, "1.0.0", permanent},  
 {erlcount, "1.0.0", transient}]}.
```

This just tells you all the same content as my manual recipe, although we can specify how we want the applications to be started (temporary, transient, permanent). We can also specify versions so we can mix and match different libraries from different Erlang versions

depending on our needs. To get all the version numbers in there, you can just do the following sequence of calls:

```
$ erl
Erlang R14B03 (erts-5.8.4) [source] [smp:2:2] [rq:2] [async-threads:0] [hipe] [kernel-poll:false]

Eshell V5.8.4 (abort with ^G)
1> application:which_applications().
[{stdlib,"ERTS CXC 138 10","1.17.4"},
 {kernel,"ERTS CXC 138 10","2.14.4"}]
```

So for that one, I was running R14B03. You can see the ERTS version in there right after the release number (the version is 5.8.4). Then by calling `application:which_applications()` on a running system, I can see the two versions I need from `kernel` (2.14.4) and `stdlib` (1.17.4). The numbers will vary from Erlang version to version. However, being explicit about the versions you need is helpful because it means that if you have many different Erlang installs, you might still only want an older version of `stdlib` that won't badly influence whatever you were doing.



You'll also note that I chose to name the `release` `erlcourt` and make it version 1.0.0. This is unrelated to the `ppool` and `erlcourt` *applications*, which are both also running the version 1.0.0, as specified in their app file.

So now we have all our applications compiled, our list of ingredients and the wonderful concept of a metaphorical *Easy-Bake Oven*®. What we need is the actual recipe.

There are a few concepts that will enter the stage right about now. A recipe will tell you a few things: in what order to add ingredients, how to mix them, how to cook them, etc. The part about the order used to add them is covered by our list of dependencies in each app file. The `systools` application will be clever enough to look at the app files and figure out what needs to run before what.

Erlang's virtual machine can start itself with a basic configuration taken from something called a *boot file*. In fact, when you start your own `erl` application from the shell, it implicitly calls the Erlang Run Time System with a default boot file. That boot file will give basic instructions such as 'load the standard library', 'load the kernel application', 'run a given

function' and so on. That boot file is a binary file created from something called a boot script, which contains tuples that will represent these instructions. We'll see how to write such a boot script.

First we start with:

```
{script, {Name, Vsn},  
 [  
 {progress, loading},  
 {preLoaded, [Mod1, Mod2, ...]},  
 {path, [Dir, "$ROOT/Dir", ...]},  
 {primLoad, [Mod1, Mod2, ...]},  
 ...
```

Just kidding. Nobody really takes the time to do that and we won't either. The boot script is something easy to generate from the .rel file. Just start an Erlang VM from the release/ directory and call the following line:

```
$ erl -env ERL_LIBS .  
...  
1> systools:make_script("erlcount-1.0", [local]).  
ok
```

Now if you look in your directory, you will have a bunch of new files, including erlcount-1.0.script and erlcount-1.0.boot files. Here, the local option means that we want the release to be possible to run from anywhere, and not just the current install. There are a bunch more options to be seen, but because systools isn't as powerful as reltool (in the next sections), we won't look into them with too much depth.

In any case, we have the boot script, but not enough to distribute our code yet. Get back to your Erlang shell and run the following command:

```
2> systools:make_tar("erlcount-1.0", [{erts, "/usr/local/lib/erlang"}]).  
ok
```

Or, on Windows 7:

```
2> systools:make_tar("erlcount-1.0", [{erts, "C:/Program Files (x86)/erl5.8.4"}]).  
ok
```

Here, systools will look for your release files and the Erlang Run Time System (because of the erts option). If you omit the erts option, the release won't be self-executable and will depend on the presence of Erlang already being installed on a system.

Running the function call above will have created an archive file named erlcount-1.0.tar.gz. Unarchive the files inside and you should see a directory like this:

```
erts-5.8.4/  
lib/  
releases/
```

The erts-5.8.4/ directory will contain the run time system. The lib/ directory holds all the applications we need and releases has the boot files and whatnot.

Move into the directory where you extracted these files. From there, we can build a command line call for erl. First of all, we specify where to find the erl executable and the boot file (without the .boot extension). In Linux, this gives us:

```
$ ./erts-5.8.4/bin/erl -boot releases/1.0.0/start
```

The command is the same for me on Windows 7, using Windows PowerShell.

Don't Drink Too Much Kool-Aid:

There is no guarantee that a release will work on any system ever. If you're using pure Erlang code, then that code will be portable. The issue is that the ERTS you ship with it might itself not work: you will either need to create many binary packages for many different platforms for large-scale definition, or just ship the BEAM files without the associated ERTS and ask people to run them with an Erlang system they have on their own computer.

You can optionally use absolute paths if you want the command to work from anywhere on your computer. Don't run it right now, though. It's going to be useless because there is no source file to analyse in the current directory. If you used absolute paths, you can go to the directory you want to analyse and call the file from there. If you used relative paths (as I did) and if you recall our erlcoun application, we made it possible to configure what directory the code will be scanning. Let's add -erlcoun directory "<path to the directory>" to the command. Then because we want this not to look like Erlang, let's add the -noshell argument. This gives me something like this on my own computer:

```
$ ./erts-5.8.4/bin/erl -boot releases/1.0.0/start -erlcoun directory "/home/ferd/code/otp_src_R14B03/" -noshell
Regex if\s+-> has 3846 results
Regex case\s+\sof has 55894 results
```

Using absolute file paths, I get something like this:

```
$ /home/ferd/code/learn-you-some-erlang/release/rel/erts-5.8.4/bin/erl -boot /home/ferd/code/learn-you-some
```

Wherever I run it from, that's the directory that's going to be scanned. Wrap this up in a shell script or a batch file and you should be good to go.

Releases With Reltool

There are a bunch of annoying things with systools. We have very little control about how things are done, and frankly, running things as they are there is a bit annoying. Manually specifying the the path to the boot file and whatnot is kind of painful. Moreover, the files are a bit large. The whole release takes over 20mb on disk, and it would be a lot worse if we were to package more applications. It is possible to do better with reltool as we get a lot more power, although the tradeoff is increased complexity.

Reltool works from a config file that looks like this:

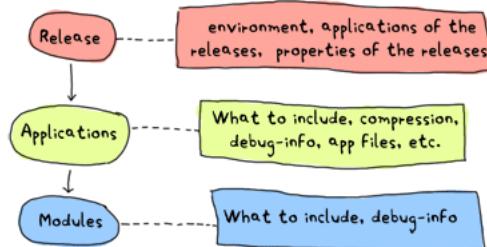
```

{sys,
 [lib_dirs, ["~/home/ferd/code/learn-you-some-erlang/release/"]],
 {rel, "erlcount", "1.0.0",
  [kernel,
   stdlib,
   {ppool, permanent},
   {erlcount, transient}
  ],
 {boot_rel, "erlcount"},
 {relocatable, true},
 {profile, standalone},
 {app, ppool, [{vsn, "1.0.0"},
   {app_file, all},
   {debug_info, keep}]}],
 {app, erlcount, [{vsn, "1.0.0"},
   {incl_cond, include},
   {app_file, strip},
   {debug_info, strip}]}]
}.

```

Behold the user friendliness of Erlang! To be quite honest, there's no easy way to introduce ourselves to Reltool. You need a bunch of these options at once or nothing will work. It might sound confusing, but there's a logic behind it.

First of all, Reltool will take different levels of information. The first level will contain release-wide information. The second level will be application-specific, before allowing fine-grained control at a module-specific level:



For each of these levels, as in the previous graph, different options will be available. Rather than taking the encyclopedic approach with all the options possible, we'll rather visit a few essential options and then a few possible configurations depending on what you might be looking for in your app.

The first option is one that helps us get rid of the somewhat annoying need to be sitting in a given directory or to set the correct `-env` arguments to the VM. The option is `lib_dirs` and it takes a list of directories where applications are sitting. So really, instead of adding `ERL_LIBS <list of directories>`, you put in `{lib_dirs, [ListOfDirectories]}` and you get the same result.

Another vital option for the Reltool configuration files is `rel`. This tuple is very similar to the `.rel` file we had written for `systools`. In the demo file above, we had:

```
{rel, "erlcount", "1.0.0",
[kernel,
stdlib,
{ppool, permanent},
{erlcount, transient}
]},
```

And that's what we'll need to tell us what apps need to be started correctly. After that tuple, we want to add a tuple of the form:

```
{boot_rel, "erlcount"}
```

This will tell Reltool that whenever someone runs the `erl` binary included in the release, we want the apps from the `erlcount` release to be started. With just these 3 options (`lib_dirs`, `rel` and `boot_rel`), we can get a valid release.

To do so, we'll put these tuples into a format Reltool can parse:

```
{sys, [
{lib_dirs, ["~/home/ferd/code/learn-you-some-erlang/release/"]},
{rel, "erlcount", "1.0.0",
[kernel,
stdlib,
{ppool, permanent},
{erlcount, transient}
]},
{boot_rel, "erlcount"}
]}.
```

Yeah, we just wrap them into a `{sys, [Options]}` tuple. In my case, I saved this in a file named `erlcount-1.0.config` in the `release/` directory. You might put it anywhere you want (except `/dev/null`, even though it's got exceptional write speeds!)

Then we'll need to open an Erlang shell:

```
1> {ok, Conf} = file:consult("erlcount-1.0.config").
{ok,[{sys,[{lib_dirs,['~/home/ferd/code/learn-you-some-erlang/release/']},
{rel,"erlcount","1.0.0",
[kernel,stdlib,{ppool,permanent},{erlcount,transient}]}],
{boot_rel,"erlcount"}]}]}
2> {ok, Spec} = reltool:get_target_spec(Conf).
{ok,[{create_dir,"releases",
...
3> reltool:eval_target_spec(Spec, code:root_dir(), "rel").
ok
```

The first step here is to read the config and bind it to the `Conf` variable. Then we send that into `reltool:get_target_spec(Conf)`. The function will take a while to run and return way too much information for us to proceed. We don't care and just save the result in `Spec`.

The third command takes the spec and tells Reltool 'I want you to take my release specification, using whatever path where my Erlang installs are, and shove it into the "rel" directory'. That's it. Look into the `rel` directory and you should have a bunch of subdirectories there.

For now we don't care and can just call:

```
$ ./bin/erl -noshell  
Regex if\s\.\+\> has 0 results  
Regex case\s\.\+\sof has 0 results
```

Ah, a bit simpler to run. You can put these files pretty much anywhere, as long as they keep the same file tree and run them from wherever you want.



Have you noticed something different? I hope you have. We didn't need to specify any version numbers. Reltool is a bit more clever than Systools there. If you do not specify a version, it will automatically look for the newest one possible in the paths you have (either in the directory returned by `code:root_dir()` or what you put in the `lib_dirs` tuple).

But what if I'm not hip and cool and trendy and all about the latest apps, but rather a retro lover? I'm still wearing my disco pants in here and I want to use older ERTS versions and older library versions, you see (I've never stayed more alive than I was in 1977!)

Thankfully, Reltool can handle releases that need to work with older versions of Erlang. Respecting your elders is an important concept for Erlang tools.

If you have older versions of Erlang installed, you can add an `{erts, [{vsn, Version}]}` entry to the config file:

```
{sys, [  
    {lib_dirs, ["~/home/ferd/code/learn-you-some-erlang/release/"]},  
    {erts, [{vsn, "5.8.3"}]},  
    {rel, "erlcount", "1.0.0",  
        [kernel,  
         stdlib,  
         {ppool, permanent},  
         {erlcount, transient}  
    }],  
},
```

```
{boot_rel, "erlcount"}  
]}.
```

Now, you want to clear out the `rel/` directory to get rid of the newer release. Then you run the rather ugly sequence of calls again:

```
4> f(),  
4> {ok, Conf} = file:consult("erlcount-1.0.config"),  
4> {ok, Spec} = reltool:get_target_spec(Conf),  
4> reltool:eval_target_spec(Spec, code:root_dir(), "rel").  
ok
```

A quick reminder here, `f()` is used to unbind the variables in the shell. Now if I go to the `rel` directory and call `$./bin/erl`, I get the following output:

```
Erlang R14B02 (erts-5.8.3) [source] ...
```

```
Eshell V5.8.3 (abort with ^G)  
1> Regex if \s.+-> has 0 results  
Regex case \s.+ \sof has 0 results
```

Awesome. This runs on version 5.8.3 even though I've got newer ones available. Ah, ha, ha, ha, Stayin' alive.

Note: if you look at the `rel/` directory, you'll see things are kind of similar to what they were with Systools, but one of the difference will be in the `lib/` directory. That one will now contain a bunch of directories and `.ez` files. The directories will contain the `include/` files required when you want to do development and the `priv/` directories when there are files that need to be kept there. The `.ez` files are just zipped beam files. The Erlang VM will unpack them for you come runtime, it's just to make things lighter.

But wait, what about my other modules?

Ah now we move away from the release-wide settings and have to enter the settings that have to do with applications. There are still a lot of release-wide options to see, but we're on such a roll that we can't be asked to stop right now. We'll revisit them in a while. For applications, we can specify versions by adding more tuples:

```
{app, AppName, [{vsn, Version}]}]
```

And put in one per app that needs it.

Now we have way more options for everything. We can specify if we want the release to include debug info, strip it away, try to make more compact app files or trust us with our definitions, stuff to include or exclude, how strict to be when it comes to including applications and modules on which your own applications might depend, etc. Moreover, these options can usually be defined both release-wide and application-wide so you can specify defaults and then values to override.

Here's a quick rundown. If you find it complex, just skip over it and you'll see a few cookbook recipes to follow:

Release-only options

{lib_dirs, [ListOfDirs]}

What directories to look inside for libraries.

{excl_lib, otp_root}

Added in R15B02, this option lets you specify OTP applications as part of your release, without including whatever comes from the standard Erlang/OTP path in the final release. This lets you create releases that are essentially libraries bootable from an existing virtual machine installed in a given system. When using this option you must now start the virtual machine as \$ erl -boot_var RELTOOL_EXT_LIB <path to release directory>/lib -boot <path to the boot file>. This will allow the release to use the current Erlang/OTP install, but with your own libraries for your custom release.

{app, AppName, [AppOptions]}

Will let you specify application-wide options, usually more specific than the release-wide options.

{boot_rel, ReleaseName}

Default release to boot with the erl executable. This means we won't need to specify the boot file when calling erl.

{rel, Name, Vsn, [Apps]}

The applications to be included in the release.

{relocatable, true | false}

It is possible to run the release from everywhere or only from a hard coded path in your system. By default it's set to true and I tend to leave it that way unless there is a good reason to do otherwise. You'll know when you need it.

{profile, development | embedded | standalone}

This option will act as a way to specify default *_filters (described below) based on your type of release. By default, development is used. That one will include more files from each app and ERTS blindly. The standalone profile will be more restrictive, and the embedded profile even more so, dropping more default ERTS applications and binaries.

Release and Application-wide Options

Note that for all of these, setting the option on the level of an application will simply override the value you gave at a system level.

{incl_sys_filters, [RegularExpressions]}

{excl_sys_filters, [RegularExpressions]}

Checks whether a file matches the include filters without matching the exclude filters before including it. You might drop or include specific files that way.

{incl_app_filters, [RegularExpressions]}

{excl_app_filters, [RegularExpressions]}

Similar to `incl_sys_filters` and `excl_sys_filters`, but for application-specific files

```
{incl_archive_filters, [RegularExpressions]}  
{excl_archive_filters, [RegularExpressions]}
```

Specified what top-level directories have to be included or excluded into `.ez` archive files (more on this soon).

```
{incl_cond, include | exclude | derived}
```

Decides how to include applications not necessarily specified in the `rel` tuple. Picking `include` means that Reltool will include pretty much everything it can find. Picking `derived` means that Reltool will only include applications that it detects can be used by any of the applications in your `rel` tuple. This is the default value. Picking `exclude` will mean that you will include no apps at all by default. You usually set this on a release-level when you want minimal includes, and then override it on an application-by-application basis for the stuff you feel like adding.

```
{mod_cond, all | app | ebin | derived | none}
```

This controls the module inclusion policy. Picking `none` means no modules will be kept. That's not very useful. The `derived` option means that Reltool will try to figure out what modules are used by other modules which are already included and add them in that case. Setting the option to `app` will mean that Reltool keeps all the modules mentioned in the `app` file and those that were derived. Setting it to `ebin` will keep those in the `ebin`/ directory and the derived ones. Using the option `all` will be a mix of using `ebin` and `app`. That's the default value.

```
{app_file, keep | strip | all}
```

This option manages how the `app` files are going to be managed for you when you include an application. Picking `keep` will guarantee that the `app` file used in the release is the same one you wrote for your application. That's the default option. If you choose `strip`, Reltool will try to generate a new `app` file that removes the modules you don't want in there (those that were excluded by filters and other options). Choosing `all` will keep the original file, but will also add specifically included modules in there. The nice thing with `all` is that it can generate `app` files for you if none are available.

Module-specific Options

```
{incl_cond, include | exclude | derived}
```

This lets you override the `mod_cond` option defined at the release level and application level.

All-levels Options

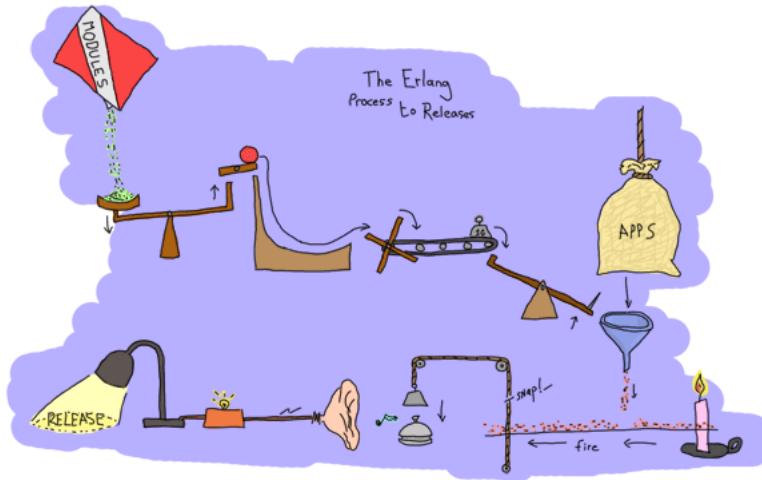
These options work on all levels. The lower the level, the more precedence it takes.

```
{debug_info, keep | strip}
```

Assuming your files were compiled with `debug_info` on (which I suggest you do), this option lets you decide whether to keep it or drop it. The `debug_info` is useful when you want to decompile files, debug them, etc. but will take some space.

THAT'S DENSE

Oh yes it is a lot of information. I didn't even cover all the possible options, but that's still a decent reference. If you want the whole thing, check out the official doc.



Recipes

For now we'll have a few general tips and tricks of things to do depending on what you want to obtain.

Development versions

Getting something for development has to be relatively easy. Often the defaults are good enough. Stick to getting the basic items we'd seen before in place and it should be enough:

```
{sys, [
  {lib_dirs, ["~/home/ferd/code/learn-you-some-erlang/release/"]},
  {rel, "erlcount", "1.0.0", [kernel, stdlib, ppool, erlcount]},
  {boot_rel, "erlcount"}
]}.
```

Reltool will take care about importing enough to be fine. In some cases, you might want to have everything from a regular VM. You might be distributing an entire VM for a team, with some libraries included. In that case, what you want to do is something more like this:

```
{sys, [
  {lib_dirs, ["~/home/ferd/code/learn-you-some-erlang/release/"]},
  {rel, "start_clean", "1.0.0", [kernel, stdlib]},
  {incl_cond, include},
  {debug_info, keep}
]}.
```

By setting `incl_cond` to `include`, all applications found in the current ERTS install and the `lib_dirs` will be part of your release.

Note: when no boot_rel is specified, you have to have a release named start_clean for reltool to be happy. That one will be picked by default when you start the associated erl file.

If we want to exclude a specific application, let's say megaco because I never looked into it, we can instead get a file like this:

```
{sys, [
  {lib_dirs, ["~/home/ferd/code/learn-you-some-erlang/release/"]},
  {rel, "start_clean", "1.0.0", [kernel, stdlib]},
  {incl_cond, include},
  {debug_info, keep},
  {app, megaco, [{incl_cond, exclude}]}
]}.
```

Here we can specify one or more applications (each having its own app tuple), and each of them overrides the incl_cond setting put at the release level. So in this case, we will include everything except megaco.

Only importing or exporting part of a library

In our release, one annoying thing that happened was that apps like ppool and others, even though they didn't need them, also kept their test files in the release. You can see them by going into rel/lib/ and unzipping ppool-1.0.0.ez (you might need to change the extension first).

To get rid of these files, the easiest way to do it is specify exclusion filters such as:

```
{sys, [
  {lib_dirs, ["~/home/ferd/code/learn-you-some-erlang/release/"]},
  {rel, "start_clean", "1.0.0", [kernel, stdlib, ppool, erlcount]},
  {excl_app_filters, ["_tests.beam$"]}
]}.
```

When you want to only import specific files of an application, let's say our erlcount_lib for its functionality, but nothing else from there, things get a bit more complex:

```
{sys, [
  {lib_dirs, ["~/home/ferd/code/learn-you-some-erlang/release/"]},
  {rel, "start_clean", "1.0.0", [kernel, stdlib]},
  {incl_cond, derived}, % exclude would also work, but not include
  {app, erlcount, [{incl_app_filters, ["^ebin/erlcount_lib.beam$"]}],
   {incl_cond, include}}
]}.
```

In this case, we switched from {incl_cond, include} to more restrictive incl_cons. This is because if you go large and rake everything is, then the only way to include a single lib is to exclude all the others with an excl_app_filters. However, when our selection is more restrictive (in this case we're derived and wouldn't include erlcount because it's not part of the rel tuple), we can specifically tell the release to include the erlcount app with only files that match the regular

expression having to do with `erlcount_lib`. This prompts the question as to how to make the most restrictive application possible, right?

Smaller Apps For Programmers With Big Hearts

This is where Reltool becomes a good bit more complex, with a rather verbose configuration file:

```
{sys, [
  {lib_dirs, ["~/home/ferd/code/learn-you-some-erlang/release/"]},
  {erts, [{mod_cond, derived},
    {app_file, strip}]},
  {rel, "erlcount", "1.0.0", [kernel, stdlib, ppool, erlcount]},
  {boot_rel, "erlcount"},
  {relocatable, true},
  {profile, embedded},
  {app_file, strip},
  {debug_info, strip},
  {incl_cond, exclude},
  {excl_app_filters, ["_tests.beam$"]},
  {app, stdlib, [{incl_cond, include}]},
  {app, kernel, [{incl_cond, include}]},
  {app, ppool, [{vsn, "1.0.0"}, {incl_cond, include}]},
  {app, erlcount, [{vsn, "1.0.0"}, {incl_cond, include}]}
]}.
```

Oh, a lot more stuff going on. We can see that in the case of `erts`, we ask for Reltool to keep only what's necessary in there. Having `mod_cond` to `derived` and `app_file` to `strip` will ask Reltool to check and only keep what's used for something else. That's why `{app_file, strip}` is also used on the release level.



The profile is set to `embedded`. If you looked at the `.ez` archives in the previous cases, they contained the source files, test directories, etc. When switching over to `embedded` only include files, binaries and the `priv/` directories are kept. I'm also removing `debug_info` from all files, even if they were compiled with it. This means we're going to lose some debugging ability, but it will reduce the size of files.

I'm still stripping away test files, and setting things so that no application is included until explicitly told to be (`{incl_cond, exclude}`). Then, I override this setting in each app I do want to include. If something's missing, Reltool will warn you, so you can try to move things around and play with settings until you get the results you want. It might involve having some

application settings with {mod_cond, derived} so that the minimal files of some applications are what is kept.

What's the difference in the end? Some of our more general releases would weigh in at over 35MB. The one described above is reduced to less than 20MB. We're shaving a good part of it. The size is still fairly large though. That's because of ERTS, which itself takes 18.5MB. If you want to, you can dig deeper and really micro manage how ERTS is built to get something smaller. You can alternatively pick some binary files in the ERTS that you know won't be used by your application: executables for scripts, remote running of Erlang, binaries from test frameworks, different running commands (Erlang with or without SMP, etc.)

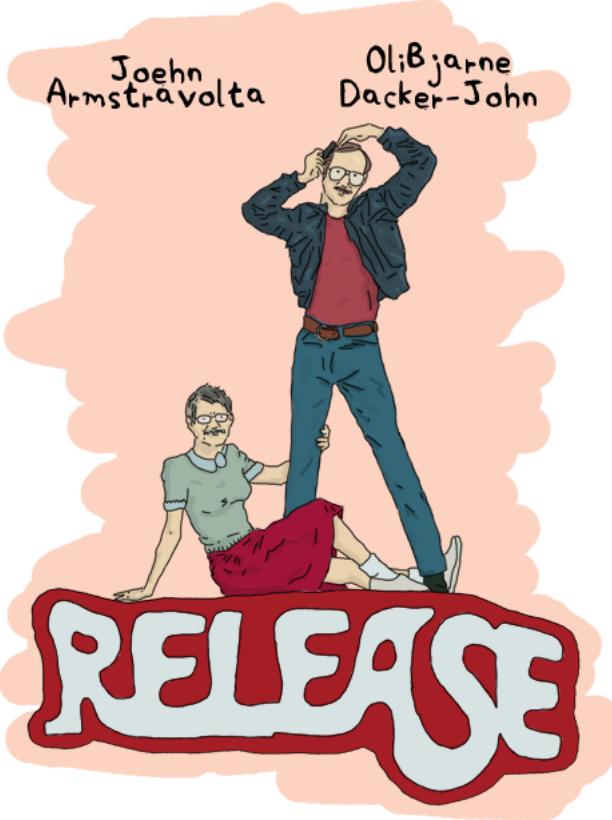
The lightest release will be the one that assumes that the other user has Erlang installed already—when you pick this option, you need to add the `rel/` directory's content as part of your `ERL_LIBS` environment variable and call the boot file yourself (a bit like with systools), but it'll work. Programmers might want to wrap this up in scripts to get things going.

Note: these days, Erlang programmers seem to really love the idea of having all these releases handled for you by a tool called `rebar3`. Rebar3 will act as a wrapper over the Erlang compiler and handle releases. There is no loss in understanding how Reltool works—Rebar3 uses a higher level abstraction for releases, and understanding reltool makes it easy to understand how any other tool works.

Released From Releases

Well, that's it for the two major ways to handle releases. It's a complex topic, but a standard way to handle things. Applications might be enough for many readers and there's nothing bad in sticking to them for a good while, but now and then releases might be useful if you want your Operations and Maintenance guy to like you a bit better given you know (or at least have some idea on) how to deploy Erlang applications when you need to.

Of course, what could make your Operations guy happier than no down time? The next challenge will be to do software upgrades while a release is running.

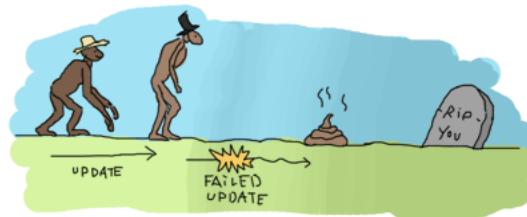


Leveling Up in The Process Quest

The Hiccups of Appups and Relups

Doing some code hot-loading is one of the simplest things in Erlang. You recompile, make a fully-qualified function call, and then enjoy. Doing it right and safe is much more difficult, though.

There is one very simple challenge that makes code reloading problematic. Let's use our amazing Erlang-programming brain and have it imagine a gen_server process. This process has a `handle_cast/2` function that accepts one kind of argument. I update it to one that takes a different kind of argument, compile it, push it in production. All is fine and dandy, but because we have an application that we don't want to shut down, we decide to load it on the production VM to make it run.



Then a bunch of error reports start pouring in. It turns out that your different `handle_cast` functions are incompatible. So when they were called a second time, no clause matched. The customer is pissed off, so is your boss. Then the operations guy is also angry because he has to get on location and rollback the code, extinguish fires, etc. If you're lucky, you're that operations guy. You're staying late and ruining the janitor's night (he usually loves to hum along with his music and dance a little bit, but he feels ashamed in your presence). You come home late, your family/friends/WoW raid party/children are mad at you, they yell, scream, slam the door and you're left alone. You had promised that nothing could go wrong, no downtime. You're using Erlang after all, right? Oh but it didn't happen so. You're left alone, curled up in a ball in the corner of the kitchen, eating a frozen hot pocket.

Of course things aren't always that bad, but the point stands. Doing live code upgrades on a production system can be very dangerous if you're changing the interface your modules give to the world: changing internal data structures, changing function names, modifying records (remember, they're tuples!), etc. They all have the potential to crash things.

When we were first playing with code reloading, we had a process with some kind of hidden message to handle doing a fully-qualified call. If you recall, a process could have looked like this:

```
loop(N) ->
    receive
        some_standard_message -> N+1;
```

```

other_message -> N-1;
{get_count, Pid} ->
    Pid ! N,
    loop(N);
update -> ?MODULE:loop(N);
end.

```

However, this way of doing things wouldn't fix our problems if we were to change the arguments to `loop/1`. We'd need to extend it a bit like this:

```

loop(N) ->
receive
    some_standard_message -> N+1;
    other_message -> N-1;
    {get_count, Pid} ->
        Pid ! N,
        loop(N);
    update -> ?MODULE:code_change(N);
end.

```

And then `code_change/1` can take care of calling a new version of `loop`. But this kind of trick couldn't work with generic loops. See this example:

```

loop(Mod, State) ->
receive
    {call, From, Msg} ->
        {reply, Reply, NewState} = Mod:handle_call(Msg, State),
        From ! Reply,
        loop(Mod, NewState);
    update ->
        {ok, NewState} = Mod:code_change(State),
        loop(Mod, NewState)
end.

```

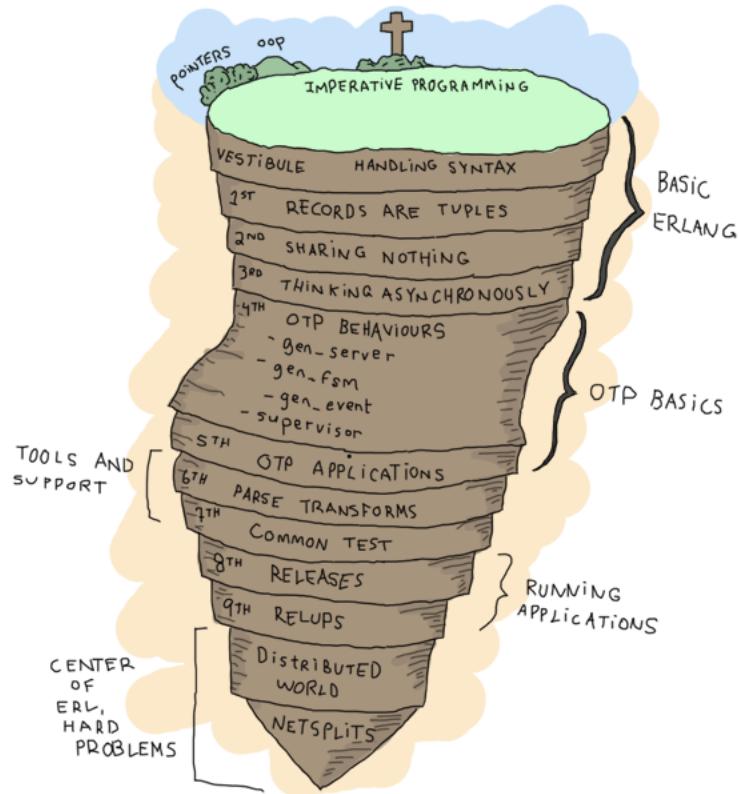
See the problem? If we want to update `Mod` and load a new version, there is *no way* to do it safely with that implementation. The call `Mod:handle_call(Msg, State)` is already fully qualified and it's well possible that a message of the form `{call, From, Msg}` is received in between the time we reload the code and handle the `update` message. In that case, we'd update the module in an uncontrolled manner. Then we'd crash.

The secret to getting it right is buried within the entrails of OTP. We must freeze the sands of time! To do so, we require more secret messages: messages to put a process on hold, messages to change the code, and then messages to resume the actions you had before. Deep inside OTP behaviours is hidden a special protocol to take care of all that kind of management. This is done through something called the `sys` module and a second one called `release_handler`, part of the SASL (System Architecture Support Libraries) application. They take care of everything.

The trick is that you can suspend OTP processes by calling `sys:suspend(PidOrName)` (you can find all of the processes using the supervision trees and looking at the children each supervisor has). Then you use `sys:change_code(PidOrName, Mod, OldVsn, Extra)` to force the process to update itself, and finally, you call `sys:resume(PidOrName)` to make things go again.

It wouldn't be very practical for us to call these functions manually by writing ad-hoc scripts all the time. Instead, we can look at how relups are done.

The 9th Circle of Erl



The act of taking a running release, making a second version of it and updating it while it runs is perilous. What seems like a simple assembly of *appups* (files containing instructions on how to update individual applications) and *relups* (file containing instructions to update an entire release) quickly turns into a struggle through APIs and undocumented assumptions.

We're getting into one of the most complex parts of OTP, difficult to comprehend and get right, on top of being time consuming. In fact, if you can avoid the whole procedure (which will be called *relup* from now on) and do simple rolling upgrades by restarting VMs and booting new applications, I would recommend you do so. Relups should be one of these 'do or die' tools. Something you use when you have few more choices.

There are a bunch of different levels to have when dealing with release upgrades:

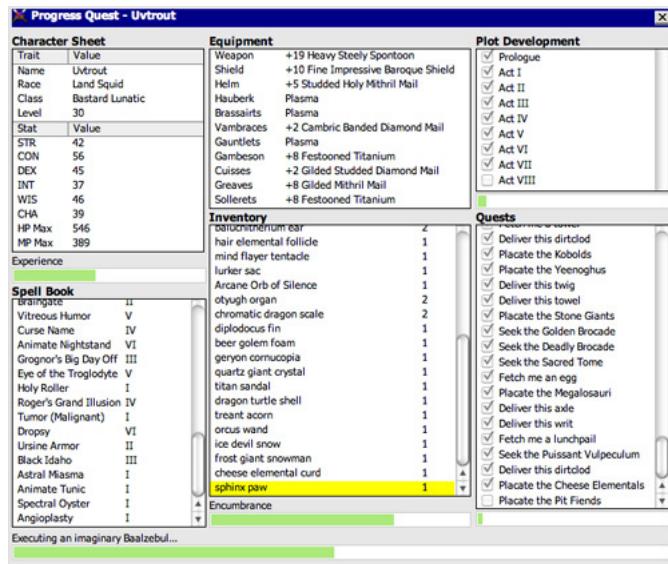
- Write OTP applications
- Turn a bunch of them into a release
- Create new versions of one or more of the OTP applications
- Create an appup file that explains what to change to make the transition between the old and the new application work
- Create a new release with the new applications
- Generate a relup file from these releases
- Install the new app in a running Erlang shell

Each of which can be more complex than the preceding one. We've only seen how to do the first 3 steps here. To be able to work with an application that is more adapted to long-running upgrades than the previous ones (eh, who cares about running regexes without restarting), we'll introduce a superb video game.

Progress Quest

Progress Quest is a revolutionary Role Playing Game. I would call it the OTP of RPGs in fact. If you've ever played an RPG before, you'll notice that many steps are similar: run around, kill enemies, gain experience, get money, level up, get skills, complete quests. Rinse and repeat forever. Power players will have shortcuts such as macros or even bots to go around and do their bidding for them.

Progress Quest took all of these generic steps and turned them into one streamlined game where all you have to do is sit back and enjoy your character doing all the work:



With the permission of the creator of this fantastic game, Eric Fredricksen, I've made a very minimal Erlang clone of it called *Process Quest*. *Process Quest* is similar in principle to *Progress Quest*, but rather than being a single-player application, it's a server able to hold many raw socket connections (usable through telnet) to let someone use a terminal and temporarily play the game.

The game is made of the following parts:

regis-1.0.0

The regis application is a process registry. It has an interface somewhat similar to the regular Erlang process registry, but it can accept any term at all and is meant to be dynamic. It might make things slower because all the calls will be serialized when they enter the server, but it will be better than using the regular process registry, which is not made for that kind of dynamic work. If this guide could automatically update itself with external libraries (it's too much work), I would have used gproc instead. It has a few modules, namely regis.erl, regis_server.erl and regis_sup.erl. The first one is a wrapper around the two other ones (and an application callback module), regis_server is the main registration gen_server, and regis_sup is the application's supervisor.

processquest-1.0.0

This is the core of the application. It includes all the game logic. Enemies, market, killing fields and statistics. The player itself is a gen_fsm that sends messages to itself in order to keep going all the time. It contains more modules than regis:

pq_enemy.erl

This module randomly picks an enemy to fight, of the form {<<"Name">>, [{drop, {<<"DropName">>, Value}}, {experience, ExpPoints}]}]. This lets the player fight an enemy.

pq_market.erl

This implements a market that allows to find items of a given value and a given strength. All items returned are of the form {<<"Name">>, Modifier, Strength, Value}. There are functions to fetch weapons, armors, shields and helmets.

pq_stats.erl

This is a small attribute generator for your character.

pq_events.erl

A wrapper around a gen_event event manager. This acts as a generic hub to which subscribers connect themselves with their own handlers to receive events from each player. It also takes care of waiting a given delay for the player's actions to avoid the game being instantaneous.

pq_player.erl

The central module. This is a gen_fsm that goes through the state loop of killing, then going to the market, then killing again, etc. It uses all of the above modules to function.

pq_sup.erl

A supervisor that sits above a pair of pq_event and pq_player processes. They both need to be together in order to work, otherwise the player process is useless and isolated or the event manager will never get any events.

pq_supersup.erl

The top-level supervisor of the application. It sits over a bunch of pq_sup processes. This lets you spawn as many players as you'd like.

processquest.erl

A wrapper and application callback module. It gives the basic interface to a player: you start one, then subscribe to events.

sockserv-1.0.0



A customized raw socket server, made to work only with the processquest app. It will spawn gen_servers each in charge of a TCP socket that will push strings to some client. Again, you may use telnet to work with it. Telnet was technically not made for raw socket connections and is its own protocol, but most modern clients accept it without a problem. Here are its modules:

sockserv_trans.erl

This translates messages received from the player's event manager into printable strings.

sockserv_pq_events.erl

A simple event handler that takes whatever events come from a player and casts them to the socket gen_server.

sockserv_serv.erl

A gen_server in charge of accepting a connection, communicating with a client and forwarding information to it.

sockserv_sup.erl

Supervises a bunch of socket servers.

sockserv.erl

Application callback module for the app as a whole.

The release

I've set everything up in a directory called processquest with the following structure:

```
apps/
- processquest-1.0.0
  - ebin/
  - src/
  - ...
- regis-1.0.0
  - ...
- sockserv-1.0.0
  - ...
rel/
```

```
(will hold releases)
processquest-1.0.0.config
```

Based on that, we can build a release.

Note: if you go look into processquest-1.0.0.config, you will see that applications such as crypto and sasl are included. Crypto is necessary to have good initialisation of pseudo-random number generators and SASL is mandatory to be able to do appups on a system. *If you forget to include SASL in your release, it will be impossible to upgrade the system*

A new filter has appeared in the config file: {excl_archive_filters, ["*"]}. This filter makes sure that no .ez file is generated, only regular files and directories. This is necessary because the tools we're going to use can not look into .ez files to find the items they need.

You will also see that there are no instructions asking to strip the debug_info. Without debug_info, doing an appup will fail for some reason.

Following last chapter's instructions, we start by calling erl -make for all applications. Once this is done, start an Erlang shell from the processquest directory and type in the following:

```
1> {ok, Conf} = file:consult("processquest-1.0.0.config"), {ok, Spec} = reltool:get_target_spec(Conf), reltool:eval_target(ok)
```

We should have a functional release. Let's try it. Start any version of the VM by doing ./rel/bin/erl -sockserv port 8888 (or any other port number you want. Default is 8082). This will show a lot of logs about processes being started (that's one of the functions of SASL), and then a regular Erlang shell. Start a telnet session on your localhost using whatever client you want:

```
$ telnet localhost 8888
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^].
What's your character's name?
hakvroot
Stats for your character:
Charisma: 7
Constitution: 12
Dexterity: 9
Intelligence: 8
Strength: 5
Wisdom: 16
```

Do you agree to these? y/n

That's a bit too much wisdom and charisma for me. I type in n then <Enter>:

```
n
Stats for your character:
Charisma: 6
```

Constitution: 12
Dexterity: 12
Intelligence: 4
Strength: 6
Wisdom: 10

Do you agree to these? y/n

Oh yes, that's ugly, dumb and weak. Exactly what I'm looking for in a hero based on me:

y
Executing a Wildcat...
Obtained Pelt.
Executing a Pig...
Obtained Bacon.
Executing a Wildcat...
Obtained Pelt.
Executing a Robot...
Obtained Chunks of Metal.
...
Executing a Ant...
Obtained Ant Egg.
Heading to the marketplace to sell loot...
Selling Ant Egg
Got 1 bucks.
Selling Goblin hair
Got 1 bucks.
...
Negotiating purchase of better equipment...
Bought a plastic knife
Heading to the killing fields...
Executing a Pig...
Obtained Bacon.
Executing a Ant...

OK, that's enough for me. Type in quit then <Enter> to close the connection:

quit
Connection closed by foreign host.

If you want, you can leave it open, see yourself level up, gain stats, etc. The game basically works, and you can try with many clients. It should keep going without a problem.

Awesome right? Well...

Making Process Quest Better



There are a few issues with the current versions of the applications of Process Quest. First of all, we have very little variety in terms of enemies to beat. Second, we have text that looks a bit weird (what is it with Executing a Ant...). A third issue is that the game is a bit too simple; let's add a mode for quests! Another one is that the value of items is directly bound to your level in the real game, while our version doesn't do it. Last of all, and you couldn't see this unless you read the code and tried to close the client on your own end, a client closing their connection will leave the player process alive on the server. Uh oh, memory leaks!

I'll have to fix this! First, I started by making a new copy of both applications that need fixes. I now have processquest-1.1.0 and sockserv-1.0.1 on top of the others (I use the version scheme of MajorVersion.Enhancements.BugFixes). Then I implemented all the changes I needed. I won't go through all of them, because the details are too many for the purpose of this chapter — we're here to upgrade an app, not to know all its little details and intricacies. In the case you do want to know all the little intricacies, I made sure to comment all of the code in a decent way so that you might be able to find the information you need to understand it. First, the changes to processquest-1.1.0. All in all, changes were brought to pq_enemy.erl, pq_events.erl, pq_player.erl and I added a file named pq_quest.erl, that implements quests based on how many enemies were killed by a player. Of these files, only pq_player.erl had changes incompatible that will require a time suspension. The change I brought was to change the record:

```
-record(state, {name, stats, exp=0, lvlexp=1000, lvl=1,  
equip=[], money=0, loot=[], bought=[], time=0}).
```

To this one:

```
-record(state, {name, stats, exp=0, lvlexp=1000, lvl=1,  
equip=[], money=0, loot=[], bought=[],  
time=0, quest}).
```

Where the quest field will hold a value given by pq_quest:fetch/0. Because of that change, I'll need to modify the code_change/4 function in the version 1.1.0. In fact I'll need to modify it twice: once in the case of an upgrade (moving from 1.0.0 to 1.1.0), and another time in the case of a downgrade (1.1.0 to 1.0.0). Fortunately, OTP will pass us different arguments in each case. When we upgrade, we get a version number for the module. We don't exactly care for

that one at this point and we'll likely just ignore it. When we downgrade, we get `{down, Version}`. This lets us easily match on each operation:

```
code_change({down, _}, StateName, State, _Extra) ->
    ...
code_change(_OldVsn, StateName, State, _Extra) ->
    ...
```

But hold on a second right there! We can't just blindly take the state as we usually do. We need to upgrade it. The problem is, we can't do something like:

```
code_change(_OldVsn, StateName, S = #state{}, _Extra) ->
    ...
```

We have two options. The first one is to declare a new state record that will have a new form. We'd end up having something like:

```
-record(state, {...}).
-record(new_state, {...}).
```

And then we'd have to change the record in each of the function clauses of the module. That's annoying and not worth the risk. It will be simpler, instead, to expand the record to its underlying tuple form (remember [A Short Visit to Common Data Structures](#)):

```
code_change({down, _},
            StateName,
            #state{name=N, stats=S, exp=E, Lvlexp=LE, Lvl=L, equip=Eq,
                   money=M, loot=Lo, bought=B, time=T},
            _Extra) ->
    Old = {state, N, S, E, LE, L, Eq, M, Lo, B, T},
    {ok, StateName, Old};
code_change(_OldVsn,
            StateName,
            {state, Name, Stats, Exp, Lvlexp, Lvl, Equip, Money, Loot,
             Bought, Time},
            _Extra) ->
    State = #state{
        name=Name, stats=Stats, exp=Exp, Lvlexp=Lvlexp, Lvl=Lvl, equip=Equip,
        money=Money, loot=Loot, bought=Bought, time=Time, quest=pq_quest:fetch()
    },
    {ok, StateName, State}.
```

And there's our `code_change/4` function! All it does is convert between both tuple forms. For new versions, we also take care of adding a new quest — it would be boring to add quests but have all our existing players unable to use them. You'll notice that we still ignore the `_Extra` variable. This one is passed from the appup file (to be described soon), and you'll be the one to pick its value. For now, we don't care because we can only upgrade and downgrade to and from one release. In some more complex cases, you might want to pass release-specific information in there.

For the `sockserv-1.0.1` application, only `sockserv_serv.erl` required changes. Fortunately, they didn't need a restart, only a new message to match on.

The two versions of the two applications have been fixed. That's not enough to go on our merry way though. We have to find a way to let OTP know what kind of changes require different kinds of actions.

Appup Files

Appup files are lists of Erlang commands that need to be done to upgrade a given application. They contain lists of tuples and atoms telling what to do and in what case. The general format for them is:

```
{NewVersion,  
 [{VersionUpgradingFrom, [Instructions]}]  
 [{VersionDownGradingTo, [Instructions]}]}.
```

They ask for lists of versions because it's possible to upgrade and downgrade to many different versions. In our case, for `processquest-1.1.0`, this would be:

```
{"1.1.0",  
 [{"1.0.0", [Instructions]}],  
 [{"1.0.0", [Instructions]}]}.
```

The instructions contain both high-level and low-level commands. We usually only need to care about high-level ones, though.

`{add_module, Mod}`

The module `Mod` is loaded for the first time.

`{load_module, Mod}`

The module `Mod` is already loaded in the VM and has been modified.

`{delete_module, Mod}`

The module `Mod` is removed from the VM.

`{update, Mod, {advanced, Extra}}`

This will suspend all processes running `Mod`, call the `code_change` function of your module with `Extra` as the last argument, then resume all processes running `Mod`. `Extra` can be used to pass in arbitrary data to the `code_change` function, in case it's required for upgrades.

`{update, Mod, supervisor}`

Calling this lets you re-define the `init` function of a supervisor to influence its restart strategy (`one_for_one`, `rest_for_one`, etc.) or change child specifications (this will not affect existing processes).

`{apply, {M, F, A}}`

Will call `apply(M,F,A)`.

Module dependencies

You can use `{load_module, Mod, [ModDependencies]}` or `{update, Mod, {advanced, Extra}, [ModDeps]}` to make sure that a command happens only after some other modules were handled beforehand. This is especially useful if `Mod` and its dependencies are *not* part of the same application. There is sadly no way to give similar dependencies to `delete_module` instructions.

Adding or removing an application

When generating relups, we won't need any special instructions to remove or add applications. The function that generates `relup` files (files to upgrade releases) will take care of detecting this for us.

Using these instructions, we can write the two following appup files for our applications. The file must be named `NameOfYourApp.appup` and be put in the app's `ebin/` directory. Here's `processquest-1.1.0`'s appup file:

```
{"1.1.0",
 [{"1.0.0", [{add_module, pq_quest},
    {load_module, pq_enemy},
    {load_module, pq_events},
    {update, pq_player, {advanced, []}, [pq_quest, pq_events]}]}],
 [{"1.0.0", [{update, pq_player, {advanced, []}},
    {delete_module, pq_quest},
    {load_module, pq_enemy},
    {load_module, pq_events}]}]}].
```

You can see that we need to add the new module, load the two ones that require no suspension, and then update `pq_player` in a safe manner. When we downgrade the code, we do the exact same thing, but in reverse. The funny thing is that in one case, `{load_module, Mod}` will load a new version, and in the other, it will load the old version. It all depends on the context between an upgrade and a downgrade.

Because `sockserv-1.0.1` had only one module to change and that it required no suspension, its appup file is only:

```
{"1.0.1",
 [{"1.0.0", [{load_module, sockserv_serv}]}],
 [{"1.0.0", [{load_module, sockserv_serv}]}]}].
```

Woo! The next step is to build a new release using the new modules. Here's the file `processquest-1.1.0.config`:

```
{sys,
  {lib_dirs, ["~/Users/ferd/code/learn-you-some-erlang/processquest/apps"]},
  {erts, [{mod_cond, derived},
    {app_file, strip}]},
  {rel, "processquest", "1.1.0",
    [kernel, stdlib, sasl, crypto, regis, processquest, sockserv]},
  {boot_rel, "processquest"},
  {relocatable, true},
  {profile, embedded},
```

```

{app_file, strip},
{incl_cond, exclude},
{excl_app_filters, ["_tests.beam"]},
{excl_archive_filters, [".*"]},
{app, stdlib, [{incl_cond, include}]},
{app, kernel, [{incl_cond, include}]},
{app, sasl, [{incl_cond, include}]},
{app, crypto, [{incl_cond, include}]},
{app, regis, [{vsn, "1.0.0"}, {incl_cond, include}]},
{app, sockserv, [{vsn, "1.0.1"}, {incl_cond, include}]},
{app, processquest, [{vsn, "1.1.0"}, {incl_cond, include}]}]
].

```

It's a copy/paste of the old one with a few versions changed. First, compile both new applications with `erl -make`. If you have downloaded the zip file earlier, they were already there for you. Then we can generate a new release. First, compile the two new applications, and then type in the following:

```
$ erl -env ERL_LIBS apps/
1> {ok, Conf} = file:consult("processquest-1.1.0.config"), {ok, Spec} = reltool:get_target_spec(Conf), reltool:eval_target
ok
```

Don't Drink Too Much Kool-Aid:

Why didn't we just use systools? Well systools has its share of issues. First of all, it will generate appup files that sometimes have weird versions in them and won't work perfectly. It will also assume a directory structure that is barely documented, but somewhat close to what reltool uses. The biggest issue, though, is that it will use your default Erlang install as the root directory, which might create all kinds of permission issues and whatnot when the time comes to unpack stuff.

There's just no easy way with either tools and we'll require a lot of manual work for that. We thus make a chain of commands that uses both modules in a rather complex manner, because it ends up being a little bit less work.

But wait, there's more manual work required!

1. `copy rel/releases/1.1.0/processquest.rel AS rel/releases/1.1.0/processquest-1.1.0.rel.`
2. `copy rel/releases/1.1.0/processquest.boot AS rel/releases/1.1.0/processquest-1.1.0.boot.`
3. `copy rel/releases/1.1.0/processquest.boot AS rel/releases/1.1.0/start.boot.`
4. `copy rel/releases/1.0.0/processquest.rel AS rel/releases/1.0.0/processquest-1.0.0.rel.`
5. `copy rel/releases/1.0.0/processquest.boot AS rel/releases/1.0.0/processquest-1.0.0.boot.`
6. `copy rel/releases/1.0.0/processquest.boot AS rel/releases/1.0.0/start.boot.`

Now we can generate the `relup` file. To do this, start an Erlang shell and call the following:

```
$ erl -env ERL_LIBS apps/ -pa apps/processquest-1.0.0/ebin/ -pa apps/sockserv-1.0.0/ebin/
1> systools:make_relup("./rel/releases/1.1.0/processquest-1.1.0", ["rel/releases/1.0.0/processquest-1.0.0"], ["rel/releases",
ok
```

Because the `ERL_LIBS` env variable will only look for the newest versions of applications, we also need to add the `-pa <Path to older applications>` in there so that systools' relup generator will be able to find everything. Once this is done, move the relup file to `rel/releases/1.1.0/`. That directory will be looked into when updating the code in order to find the right stuff in there. One problem we'll have, though, is that the release handler module will depend on a bunch of files it assumes to be present, but won't necessarily be there.



Upgrading the Release

Sweet, we've got a relup file. There's still stuff to do before being able to use it though. The next step is to generate a tar file for the whole new version of the release:

```
2> systools:make_tar("rel/releases/1.1.0/processquest-1.1.0").  
ok
```

The file will be in `rel/releases/1.1.0/`. We now need to manually move it to `rel/releases`, and rename it to add the version number when doing so. More hard-coded junk! `$ mv rel/releases/1.1.0/processquest-1.1.0.tar.gz rel/releases/` is our way out of this.

Now this is a step you want to do at *any time before you start the real production application*. This is a step that needs to be done *before* you start the application as it will allow you to rollback to the initial version after a relup. If you do not do this, you will be able to downgrade production applications only to releases newer than the first one, but not the first one!

Open a shell and run this:

```
1> release_handler:create_RELEASES("rel", "rel/releases", "rel/releases/1.0.0/processquest-1.0.0.rel", [{kernel,"2.14.4", "rel/
```

The general format of the function is `release_handler:create_RELEASES(RootDir, ReleasesDir, Relfile, [{AppName, Vsn, LibDir}])`. This will create a file named `RELEASES` inside the `rel/releases` directory (or any other `ReleasesDir`) that will contain basic information on your releases when relup is looking for files and modules to reload.

We can now start running the old version of the code. If you start `rel/bin/erl`, it will start the 1.1.0 release by default. That's because we built the new release before starting the VM. For this demonstration, we'll need to start the release with `./rel/bin/erl -boot rel/releases/1.0.0/processquest`. You should see everything starting up. Start a telnet client to connect to our socket server so we can see the live upgrade taking place.

Whenever you feel ready for an upgrade, go to the Erlang shell currently running ProcessQuest, and call the following function:

```
1> release_handler:unpack_release("processquest-1.1.0").  
{ok,"1.1.0"}  
2> release_handler:which_releases().  
[{"processquest","1.1.0",  
 ["kernel-2.14.4","stdlib-1.17.4","crypto-2.0.3",  
 "regis-1.0.0","processquest-1.1.0","sockserv-1.0.1",  
 "sasl-2.1.9.4"],  
 unpacked},  
 {"processquest","1.0.0",  
 ["kernel-2.14.4","stdlib-1.17.4","crypto-2.0.3",  
 "regis-1.0.0","processquest-1.0.0","sockserv-1.0.0",  
 "sasl-2.1.9.4"],  
 permanent}]
```

The second prompt here tells you that the release is ready to be upgraded, but not installed nor made permanent yet. To install it, do:

```
3> release_handler:install_release("1.1.0").  
{ok,"1.0.0",[]}  
4> release_handler:which_releases().  
[{"processquest","1.1.0",  
 ["kernel-2.14.4","stdlib-1.17.4","crypto-2.0.3",  
 "regis-1.0.0","processquest-1.1.0","sockserv-1.0.1",  
 "sasl-2.1.9.4"],  
 current},  
 {"processquest","1.0.0",  
 ["kernel-2.14.4","stdlib-1.17.4","crypto-2.0.3",  
 "regis-1.0.0","processquest-1.0.0","sockserv-1.0.0",  
 "sasl-2.1.9.4"],  
 permanent}]
```

So now, the release 1.1.0 should be running, but it's still not there forever. Still, you could keep your application just running that way. Call the following function to make things permanent:

```
5> release_handler:make_permanent("1.1.0").  
ok.
```

Ah damn. A bunch of our processes are dying now (error output removed from the sample above). Except that if you look at our telnet client, it did seem to upgrade fine. The issue is that all the gen_servers that were waiting for connections in sockserv could not listen to messages because accepting a TCP connection is a blocking operation. Thus, the servers couldn't upgrade when new versions of the code were loaded and were killed by the VM. See how we can confirm this:

```
6> supervisor:which_children(sockserv_sup).  
[{undefined,<0.51.0>,worker,[sockserv_serv]}]
```

```

7> [sockserv_sup:start_socket() || _ <- lists:seq(1,20)].
[{:ok,<0.99.0>},
{:ok,<0.100.0>},
...
{:ok,<0.117.0>},
{:ok,<0.118.0>}]
8> supervisor:which_children(sockserv_sup).
[{undefined,<0.112.0>,worker,[sockserv_serv]},
{undefined,<0.113.0>,worker,[sockserv_serv]},
...
{undefined,<0.109.0>,worker,[sockserv_serv]},
{undefined,<0.110.0>,worker,[sockserv_serv]},
{undefined,<0.111.0>,worker,[sockserv_serv]}]

```

The first command shows that all children that were waiting for connections have already died. The processes left will be those with an active session going on. This shows the importance of keeping code responsive. Had our processes been able to receive messages and act on them, things would have been fine.



In the two last commands, I just start more workers to fix the problem. While this works, it requires manual action from the person running the upgrade. In any case, this is far from optimal. A better way to solve the problem would be to change the way our application works in order to have a monitor process watching how many children `sockserv_sup` has. When the number of children falls under a given threshold, the monitor starts more of them. Another strategy would be to change the code so accepting connections is done by blocking on intervals of a few seconds at a time, and keep retrying after pauses where messages can be received. This would give the `gen_servers` the time to upgrade themselves as required, assuming you'd wait the right delay between the installation of a release and making it permanent. Implementing either or both of these solutions is left as an exercise to the reader because I am somewhat lazy. These kinds of crashes are the reason why you want to test your code before doing these updates on a live system.

In any case, we've solved the problem for now and we might want to check how the upgrade procedure went:

```

9> release_handler:which_releases().
[{"processquest","1.1.0",
["kernel-2.14.4","stdlib-1.17.4","crypto-2.0.3",
"regis-1.0.0","processquest-1.1.0","sockserv-1.0.1",
"sasl-2.1.9.4"],

```

```

permanent],
{"processquest","1.0.0",
["kernel-2.14.4","stdlib-1.17.4","crypto-2.0.3",
"regis-1.0.0","processquest-1.0.0","sockserv-1.0.0",
"sasl-2.1.9.4"],
old}]

```

That's worth a fist pump. You can try downgrading an installation by doing `release_handler:install(OldVersion)`. This should work fine, although it could risk killing more processes that never updated themselves.

Don't Drink Too Much Kool-Aid:

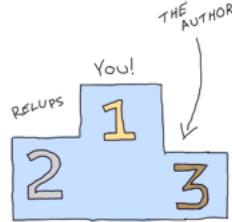
If for some reason, rolling back always fails when trying to roll back to the first version of the release using the techniques shown in this chapter, you have probably forgotten to create the RELEASES file. You can know this if you see an empty list in `{YourRelease,Version,[],Status}` when calling `release_handler:which_releases()`. This is a list of where to find modules to load and reload, and it is first built when booting the VM and reading the RELEASES file, or when unpacking a new release.

Ok, so here's a list of all the actions that must be taken to have functional relups:

1. Write OTP applications for your first software iteration
2. Compile them
3. Build a release (1.0.0) using Reltool. It must have debug info and no .ez archive.
4. Make sure you create the RELEASES file at some point before starting your production application. You can do it with `release_handler:create_RELEASES(RootDir, ReleasesDir, Relfile, [{AppName, Vsn, LibDir}])`.
5. Run the release!
6. Find bugs in it
7. Fix bugs in new versions of applications
8. Write appup files for each of the applications
9. Compile the new applications
10. Build a new release (1.1.0 in our case). It must have debug info and no .ez archive
11. Copy `rel/releases/NewVsn/RelName.rel` as `rel/releases/NewVsn/RelName-NewVsn.rel`
12. Copy `rel/releases/NewVsn/RelName.boot` as `rel/releases/NewVsn/RelName-NewVsn.boot`
13. Copy `rel/releases/NewVsn/RelName.boot` as `rel/releases/NewVsn/start.boot`
14. Copy `rel/releases/OldVsn/RelName.rel` as `rel/releases/OldVsn/RelName-OldVsn.rel`
15. Copy `rel/releases/OldVsn/RelName.boot` as `rel/releases/OldVsn/RelName-OldVsn.boot`
16. Copy `rel/releases/OldVsn/RelName.boot` as `rel/releases/OldVsn/start.boot`
17. Generate a relup file with `systools:make_relup("rel/releases/Vsn/RelName-Vsn", ["rel/releases/OldVsn/RelName-OldVsn"], ["rel/releases/DownVsn/RelName-DownVsn"])`.
18. Move the relup file to `rel/releases/Vsn`
19. Generate a tar file of the new release with `systools:make_tar("rel/releases/Vsn/RelName-Vsn")`.
20. Move the tar file to `rel/releases/`
21. Have some shell opened that still runs the first version of the release

22. Call `release_handler:unpack_release("NameOfRel-Vsn")`.
23. Call `release_handler:install_release(Vsn)`.
24. Call `release_handler:make_permanent(Vsn)`.
25. Make sure things went fine. If not, rollback by installing an older version.

You might want to write a few scripts to automate this.

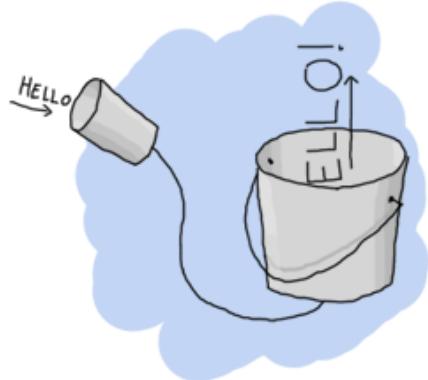


Again, relups are a very messy part of OTP, a part that is hard to grasp. You will likely find yourself finding plenty of new errors, which are all more impossible to understand than the previous ones. Some assumptions are made about how you're going to run things, and choosing different tools when creating releases will change how things should be done. You might be tempted to write your own update code using the `sys` module's functions even! Or maybe use tools like `rebar3` which will automate some of the painful steps. In any case, this chapter and its examples have been written to the best knowledge of the author, a person who sometimes enjoys writing about himself in third person.

If it is possible to upgrade your application in ways that do not require relups, I would recommend doing so. It is said that divisions of Ericsson that do use relups spend as much time testing them as they do testing their applications themselves. They are a tool to be used when working with products that can imperatively never be shut down. You will know when you will need them, mostly because you'll be ready to go through the hassle of using them (got to love that circular logic!) When the need arises, relups are entirely useful.

How about we go learn about some friendlier features of Erlang, now?

Buckets of Sockets



So far we've had some fun dealing with Erlang itself, barely communicating to the outside world, if only by text files that we read here and there. As much of relationships with yourself might be fun, it's time to get out of our lair and start talking to the rest of the world.

This chapter will cover three components of using sockets: IO lists, UDP sockets and TCP sockets. IO lists aren't extremely complex as a topic. They're just a clever way to efficiently build strings to be sent over sockets and other Erlang drivers.

IO Lists

I've mentioned earlier in this guide that for text, we could use either strings (lists of integers) or binaries (a binary data structure holding data). Sending things over the wire such as "Hello World" can be done as a string as "Hello World", and as a binary as <<"Hello World">>. Similar notation, similar results.

The difference lies in how you can assemble things. A string is a bit like a linked list of integers: for each character, you've got to store the character itself plus a link towards the rest of the list. Moreover, if you want to add elements to a list, either in the middle or at the end, you

have to traverse the whole list up to the point you're modifying and then add your elements. This isn't the case when you prepend, however:

```
A = [a]  
B = [b|A] = [b,a]  
C = [c|B] = [c,b,a]
```

In the case of prepending, as above, whatever is held into *A* or *B* or *C* never needs to be rewritten. The representation of *C* can be seen as either [c,b,a], [c|B] or [c,[b|[a]]], among others. In the last case, you can see that the shape of *A* is the same at the end of the list as when it was declared. Similarly for *B*. Here's how it looks with appending:

```
A = [a]  
B = A ++ [b] = [a] ++ [b] = [a|[b]]  
C = B ++ [c] = [a|[b]] ++ [c] = [a|[b|[c]]]
```

Do you see all that rewriting? When we create *B*, we have to rewrite *A*. When we write *C*, we have to rewrite *B* (including the [a...] part it contains). If we were to add *D* in a similar manner, we would need to rewrite *C*. Over long strings, this becomes way too inefficient, and it creates a lot of garbage left to be cleaned up by the Erlang VM.

With binaries, things are not exactly as bad:

```
A = <<"a">>  
B = <<A/binary, "b">> = <<"ab">>  
C = <<B/binary, "c">> = <<"abc">>
```

In this case, binaries know their own length and data can be joined in constant time. That's good, much better than lists. They're also more compact. For these reasons, we'll often try to stick to binaries when using text in the future.

There are a few downsides, however. Binaries were meant to handle things in certain ways, and there is still a cost to modifying binaries, splitting them, etc. Moreover, sometimes we'll work with code that uses strings, binaries, and individual characters interchangeably. Constantly converting between types would be a hassle.

In these cases, *IO lists* are our saviour. IO lists are a weird type of data structure. They are lists of either bytes (integers from 0 to 255), binaries, or other IO lists. This means that functions that accept IO lists can accept items such as `[$H, $e, [$I, <<"lo">>, " "], [[["w", "o"], <<"rl">>]] | <<"d">>]`. When this happens, the Erlang VM will just flatten the list as it needs to do it to obtain the sequence of characters Hello World.

What are the functions that accept such IO Lists? Most of the functions that have to do with outputting data do. Any function from the io module, file module, TCP and UDP sockets will be able to handle them. Some library functions, such as some coming from the unicode module and all of the functions from the re (for regular expressions) module will also handle them, to name a few.

Try the previous Hello World IO List in the shell with `io:format("~s~n", [IoList])` just to see. It should work without a problem.



All in all, they're a pretty clever way of building strings to avoid the problems of immutable data structures when it comes to

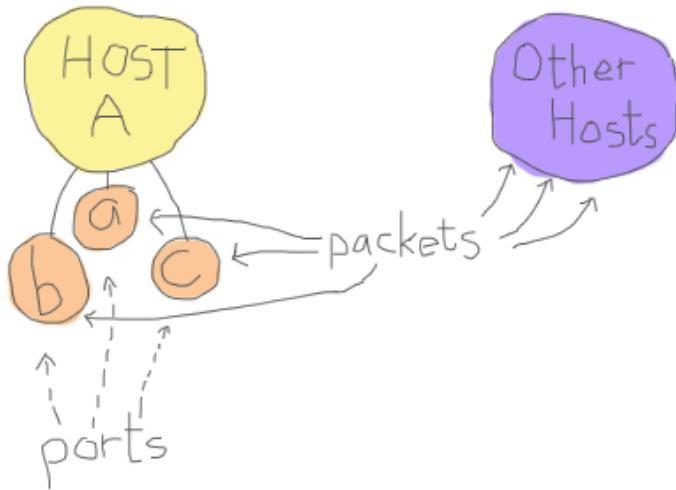
dynamically building content to be output.

TCP and UDP: Bro-tocols

The first kind of socket that we can use in Erlang is based on the UDP protocol. UDP is a protocol built on top of the IP layer that provides a few abstractions on top of it such as port numbers. UDP is said to be a stateless protocol. The data that is received from UDP port is broken in small parts, untagged, without a session, and there is no guarantee that the fragments you received were sent in the same order as you got them. In fact, there is no guarantee that if someone sends a packet, you'll receive it at all. For these reasons, people tend to use UDP when the packets are small, can sometimes be lost with little consequences, when there aren't too many complex exchanges taking place or when low latency is absolutely necessary.

This is something to be seen in opposition to stateful protocols like TCP, where the protocol takes care of handling lost packets, re-ordering them, maintaining isolated sessions between multiple senders and receivers, etc. TCP will allow reliable exchange of information, but will risk being slower and heavier to set up. UDP will be fast, but less reliable. Choose carefully depending on what you need.

In any case, using UDP in Erlang is relatively simple. We set up a socket over a given port, and that socket can both send and receive data:



For a bad analogy, this is like having a bunch of mailboxes on your house (each mailbox being a port) and receiving tiny slips of paper in each of them with small messages. They can have any content, from "I like how you look in these pants" down to "The slip is coming from *inside* the house!". When some messages are too large for a slip of paper, then many of them are dropped in the mailbox. It's your job to reassemble them in a way that makes sense, then drive up to some house, and drop slips after that as a reply. If the messages are purely informative ("hey there, your door is unlocked") or very tiny ("What are you wearing? -Ron"), it should be fine and you could use one mailbox for all of the queries. If they were to be complex, though, we might want to use one port per session, right? Ugh, no! Use TCP!

In the case of TCP, the protocol is said to be stateful, connection-based. Before being able to send messages, you have to do a handshake. This means that someone's taking a mailbox (similar to what we have in the UDP analogy), and sends a message saying 'hey dude, this is IP 94.25.12.37 calling. Wanna chat?', to which you reply something a bit similar to 'Sure. Tag your messages with number N and then add an increasing number to them'. From that point on, when you or IP 92.25.12.37 want to communicate with each other, it'll

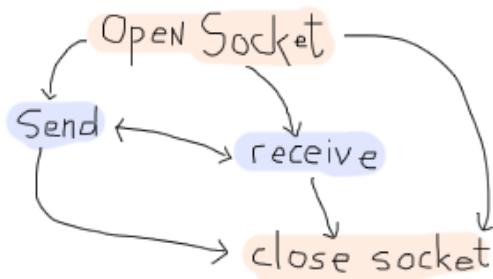
be possible to order slips of paper, ask for missing ones, reply to them and so on in a meaningful manner.

That way, we can use a single mailbox (or port) and keep all our communications fine. That's the neat thing of TCP. It adds some overhead, but makes sure that everything is ordered, properly delivered, and so on.

If you're not a fan of these analogies, do not despair because we'll cut to the chase by seeing how to use TCP and UDP sockets with Erlang right now. This should be simpler.

UDP Sockets

There are only a few basic operations with UDP: setting up a socket, sending messages, receiving messages and closing a connection. The possibilities are a bit like this:



The first operation, no matter what, is to open a socket. This is done by calling `gen_udp:open/1-2`. The simplest form is done by calling `{ok, socket} = gen_udp:open(PortNumber)`.

The port number will be any integer between 1 and 65535. From 0 to 1023, the ports are known as *system ports*. Most of the time, your operating system will make it impossible to listen to a system port unless you have administrative rights. Ports from 1024 through 49151 are registered ports. They usually require no permissions and are free

to use, although some of them are registered to well known services. Then the rest of the ports are known as *dynamic* or *private*. They're frequently used for *ephemeral ports*. For our tests, we'll take port numbers that are somewhat safe, such as 8789, unlikely to be taken.

But before that, what about `gen_udp:open/2`? The second argument can be a list of options, specifying in what type we want to receive data (`list` or `binary`), how we want them received; as messages (`{active, true}`) or as results of a function call (`{active, false}`). There are more options such as whether the socket should be set with IPv4 (`inet4`) or IPv6 (`inet6`), whether the UDP socket can be used to broadcast information (`{broadcast, true | false}`), the size of buffers, etc. There are more options available, but we'll stick to the simple stuff for now because understanding the rest is rather up to you to learn. The topic can become complex fast and this guide is about Erlang, not TCP and UDP, unfortunately.

So let's open a socket. First start a given Erlang shell:

```
1> {ok, Socket} = gen_udp:open(8789, [binary, {active,true}]).  
{ok,#Port<0.676>}  
2> gen_udp:open(8789, [binary, {active,true}]).  
{error,eaddrinuse}
```

In the first command, I open the socket, order it to return me binary data, and I want it to be active. You can see a new data structure being returned: `#Port<0.676>`. This is the representation of the socket we have just opened. They can be used a lot like Pids: you can even set up links to them so that failure is propagated to the sockets in case of a crash! The second function call tries to open the same socket over again, which is impossible. That's why `{error, eaddrinuse}` is returned. Fortunately, the first `Socket` socket is still open.

In any case, we'll start a second Erlang shell. In that one we'll open a second UDP socket, with a different port number:

```
1> {ok, Socket} = gen_udp:open(8790).
{ok,#Port<0.587>}
2> gen_udp:send(Socket, {127,0,0,1}, 8789, "hey there!").
ok
```

Ah, a new function! In the second call, `gen_udp:send/4` is used to send messages (what a wonderfully descriptive name). The arguments are, in order: `gen_udp:send(OwnSocket, RemoteAddress, RemotePort, Message)`. The `RemoteAddress` can be either a string or an atom containing a domain name ("example.org"), a 4-tuple describing an IPv4 address or a 8-tuple describing an IPv6 address. Then we specify the receiver's port number (in what mailbox are we going to drop our slip of paper?), and then the message, which can be a string, a binary, or an IO list.

Did the message ever get sent? Go back to your first shell and try to flush the data:

```
3> flush().
shell got {udp,#Port<0.676>,{127,0,0,1},8790,<<"hey there!">>}
ok
```

Fantastic. The process that opened the socket will receive messages of the form `{udp, Socket, FromIp, FromPort, Message}`. Using these fields, we'll be able to know where a message is from, what socket it went through, and what the contents were. So we've covered opening sockets, sending data, and receiving it in an active mode. What about passive mode? For this, we need to close the socket from the first shell and open a new one:

```
4> gen_udp:close(Socket).
ok
5> f(Socket).
ok
6> {ok, Socket} = gen_udp:open(8789, [binary, {active,false}]).
```

{ok,#Port<0.683>}

So here, we close the socket, unbind the `Socket` variable, then bind it as we open a socket again, in passive mode this time. Before sending a message back, try the following:

```
7> gen_udp:recv(Socket, 0).
```

And your shell should be stuck. The function here is `recv/2`. This is the function used to poll a passive socket for messages. The `0` here is the length of the message we want. The funny thing is that the length is completely ignored with `gen_udp`. `gen_tcp` has a similar function, and in that case, it does have an impact. Anyway, if we never send a message, `recv/2` is never going to return. Get back to the second shell and send a new message:

```
3> gen_udp:send(Socket, {127,0,0,1}, 8789, "hey there!").  
ok
```

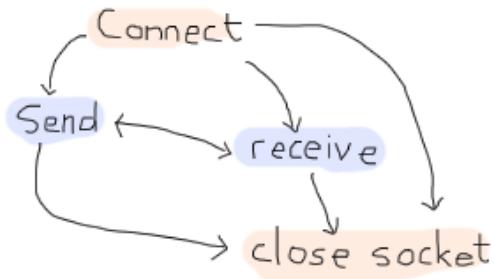
Then the first shell should have printed `{ok,{127,0,0,1},8790,<<"hey there!">>}` as the return value. What if you don't want to wait forever? Just add a time out value:

```
8> gen_udp:recv(Socket, 0, 2000).  
{error,timeout}
```

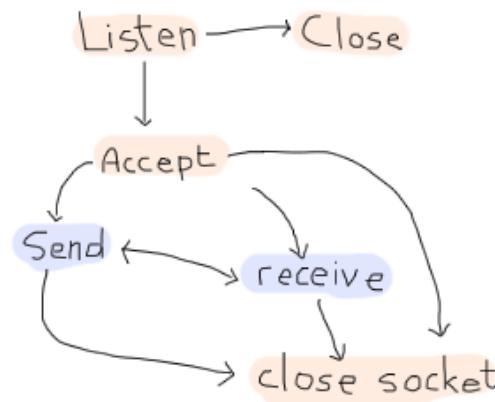
And that's most of it for UDP. No, really!

TCP Sockets

While TCP sockets share a large part of their interface with UDP sockets, there are some vital differences in how they work. The biggest one is that clients and servers are two entirely different things. A client will behave with the following operations:



While a server will rather follow this scheme:



Weird looking, huh? The client acts a bit like what we had with `gen_udp`: you connect to a port, send and receive, stop doing so. When serving, however, we have one new mode there: listening. That's because of how TCP works to set sessions up.

First of all, we open a new shell and start something called a *listen socket* with `gen_tcp:listen(Port, Options)`:

```
1> {ok, ListenSocket} = gen_tcp:listen(8091, [{active,true}, binary]).  
{ok,#Port<0.661>}
```

The listen socket is just in charge of waiting for connection requests. You can see that I used similar options as I did with `gen_udp`. That's because most options are going to be similar for all IP sockets. The TCP ones do have a few more specific options, including a connection backlog (`{backlog, N}`), keepalive sockets (`{keepalive, true}` |

`false} }, packet packaging ({packet, N}, where N is the length of each packet's header to be stripped and parsed for you), etc.`

Once the listen socket is open, any process (and more than one) can take the listen socket and fall into an 'accepting' state, locked up until some client asks to talk with it:

```
2> {ok, AcceptSocket} = gen_tcp:accept(ListenSocket, 2000).  
** exception error: no match of right hand side value {error,timeout}  
3> {ok, AcceptSocket} = gen_tcp:accept(ListenSocket).  
** exception error: no match of right hand side value {error,closed}
```

Damn. We timed out and then crashed. The listen socket got closed when the shell process it was associated with disappeared. Let's start over again, this time without the 2 seconds (2000 milliseconds) timeout:

```
4> f().  
ok  
5> {ok, ListenSocket} = gen_tcp:listen(8091, [{active, true}, binary]).  
{ok,#Port<0.728>}  
6> {ok, AcceptSocket} = gen_tcp:accept(ListenSocket).
```

And then the process is locked. Great! Let's open a second shell:

```
1> {ok, Socket} = gen_tcp:connect([{127,0,0,1}], 8091, [binary, {active,true}]).  
{ok,#Port<0.596>}
```

This one still takes the same options as usual, and you can add a *Timeout* argument in the last position if you don't want to wait forever. If you look back to the first shell, it should have returned with `{ok, SocketNumber}`. From that point on, the accept socket and the client socket can communicate on a one-on-one basis, similarly to `gen_udp`. Take the second shell and send messages to the first one:

```
3> gen_tcp:send(Socket, "Hey there first shell!").  
ok
```

And from the first shell:

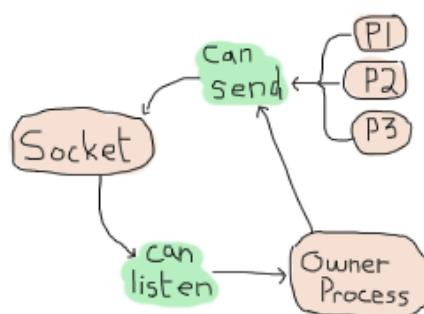
```
7> flush().  
shell got {tcp,#Port<0.729>,<<"Hey there first shell!">>}  
ok
```

Both sockets can send messages in the same way, and can then be closed with `gen_tcp:close(Socket)`. Note that closing an accept socket will close that socket alone, and closing a listen socket will close none of the related and established accept sockets, but will interrupt currently running accept calls by returning `{error, closed}`.

That's it for most of TCP sockets in Erlang! But is it really?

Ah yes, of course, there is more that can be done. If you've experimented with sockets a bit on your own, you might have noticed that there is some kind of ownership to sockets.

By this, I mean that UDP sockets, TCP client sockets and TCP accept sockets can all have messages sent through them from any process in existence, but messages received can only be read by the process that started the socket:



That's not very practical now, is it? It means that we have to always keep the owner process alive to relay messages, even if it has nothing to do with our needs. Wouldn't it be neat to be able to do something like this?

1. Process A starts a socket
2. Process A sends a request
3. Process A spawns process B
with a socket
- 4a. Gives ownership of the
socket to Process B
- 4b. Process B handles the request
- 5a. Process A sends a request
- 5b. Process B Keeps handling
the request
- 6a. Process A spawns process C
- 6b. ...
with a socket
- ...

Here, A would be in charge of running a bunch of queries, but each new process would take charge of waiting for the reply, processing it and whatnot. Because of this, it would be clever for A to delegate a new process to run the task. The tricky part here is giving away the ownership of the socket.

Here's the trick. Both `gen_tcp` and `gen_udp` contain a function called `controlling_process(Socket, Pid)`. This function has to be called by the current socket owner. Then the process tells Erlang 'you know what? Just let this *Pid* guy take over my socket. I give up'. From now on, the *Pid* in the function is the one that can read and receive messages from the socket. That's it.

More Control With Inet

So now we understand how to open sockets, send messages through them, change ownership, and so on. We also know how to listen to messages both in passive and active mode. Back in the UDP example, when I wanted to switch from active to passive mode, I restarted the socket, flushed variables and went on. This is rather unpractical, especially when we desire to do the same while using TCP because we'd have to break an active session.

Fortunately, there's a module named `inet` that takes care of handling all operations that can be common to both `gen_tcp` and `gen_udp` sockets. For our problem at hand, which was changing between active and passive modes, there's a function named `inet:setopts(Socket, Options)`. The option list can contain any terms used at the setup of a socket.

Note: be careful! There exists a module named `inet` and a module named `inets`. `inet` is the module we want here. `inets` is an OTP application that contains a bunch of pre-written services and servers (including FTP, Trivial FTP (TFTP), HTTP, etc.)

An easy trick to differentiate them is that `inets` is about **s**ervices built on top of `inet`, or if you prefer, `inet + s(ervices)`.

Start a shell to be a TCP server:

```
1> {ok, Listen} = gen_tcp:listen(8088, [{active,false}]).  
{ok,#Port<0.597>}  
2> {ok, Accept} = gen_tcp:accept(Listen).
```

And in a second shell:

```
1> {ok, Socket} = gen_tcp:connect({127,0,0,1}, 8088, []).  
{ok,#Port<0.596>}  
2> gen_tcp:send(Socket, "hey there").  
ok
```

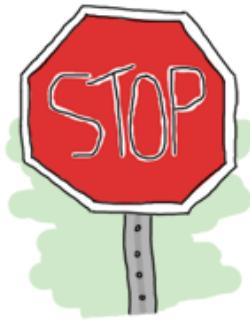
Then back to the first shell, the socket should have been accepted. We flush to see if we got anything:

```
3> flush().  
ok
```

Of course not, we're in passive mode. Let's fix this:

```
4> inet:setopts(Accept, [{active, true}]).  
ok  
5> flush().  
shell got {tcp,#Port<0.598>,"hey there"}  
ok
```

Yes! With full control over active and passive sockets, the power is ours. How do we pick between active and passive modes?



Well there are many points. In general, if you're waiting for a message right away, passive mode will be much faster. Erlang won't have to toy with your process' mailbox to handle things, you won't have to scan said mailbox, fetch messages, etc. Using `recv` will be more efficient. However, `recv` changes your process from something event-driven to active polling – if you've got to play middle-man between a socket and some other Erlang code, this might make things a bit complex.

In that case, switching to active mode will be a good idea. If packets are sent as messages, you just have to wait in a receive (or a `gen_server`'s `handle_info` function) and play with messages. The downside of this, apart from speed, has to do with rate limiting.

The idea is that if all packets coming from the outside world are blindly accepted by Erlang and then converted to messages, it is somewhat easy for someone outside of the VM to flood it and kill it. Passive mode has the advantage of restricting how and when

messages can be put into the Erlang VM, and delegating the task of blocking, queuing up, and dropping messages to the lower-level implementations.

So what if we need active mode for the semantics, but passive mode for the safety? We could try to quickly switch between passive and active with `inet:setopts/2`, but that would be rather risky for race conditions. Instead, there's a mode called *active once*, with the option `{active, once}`. Let's try it to see how it works.

Keep the shell with the server from earlier:

```
6> inet:setopts(Accept, [{active, once}]).  
ok
```

Now get to the client shell and run two more `send/2` calls:

```
3> gen_tcp:send(Socket, "one").  
ok  
4> gen_tcp:send(Socket, "two").  
ok
```

And back to server shell:

```
7> flush().  
Shell got {tcp,#Port<0.598>,"one"}  
ok  
8> flush().  
ok  
9> inet:setopts(Accept, [{active, once}]).  
ok  
10> flush().  
Shell got {tcp,#Port<0.598>,"two"}  
ok
```

See? Until we ask for `{active, once}` a second time, the message "two" hasn't been converted to a message, which means the socket was back to passive mode. So the active once mode allows us to do that

back-and-forth switch between active and passive in a safe way.
Nice semantics, plus the safety.

There are other nice functions part of inet. Stuff to read statistics, get current host information, inspect sockets and so on.

Well that's most of it for sockets. Now's time to put this into practice.

Note: out in the wilderness of the Internet, you have libraries to do so with a truckload of protocols: HTTP, Omq, raw unix sockets, etc. They're all available. The standard Erlang distribution, however, comes with two main options, TCP and UDP sockets. It also comes with some HTTP servers and parsing code, but it's not the most efficient thing around.

Update:

Starting with version 17.0, it is now possible to tell a port to be active for N packets. The `{active, N}` option for TCP and UDP ports has been added, where N can be any value from 0 to 32767. Once the remaining message counter either reaches 0 or is explicitly set to 0 through `inet:setopts/2`, the socket transitions to passive (`{active, false}`) mode. At that point, a message is sent to the socket's controlling process to inform it of the transition. The message will be `{tcp_passive, Socket}`, and `{udp_passive, Socket}` for UDP.

When calling the function multiple times, each new value is added to the total counter. Calling it with `{active, 3}` three times will have it send up to 9 messages to the controlling process. The N value can also be negative to force decrementing the counter. If the final value would be below 0, Erlang silently sets it to 0 and transitions to passive mode.



Sockserv, Revisited

I won't be introducing that much new code for this chapter. Instead, we'll look back at the sockserv server from Process Quest, in the last chapter. It's a perfectly viable server and we'll see how to deal with serving TCP connections within an OTP supervision trees, in a gen_server.

A naive implementation of a TCP server might look a bit like this:

```
-module(naive_tcp).
-compile(export_all).

start_server(Port) ->
    Pid = spawn_link(fun() ->
        {ok, Listen} = gen_tcp:listen(Port, [binary, {active, false}]),
        spawn(fun() -> acceptor(Listen) end),
        timer:sleep(infinity)
    end),
    {ok, Pid}.

acceptor(ListenSocket) ->
    {ok, Socket} = gen_tcp:accept(ListenSocket),
    spawn(fun() -> acceptor(ListenSocket) end),
    handle(Socket).

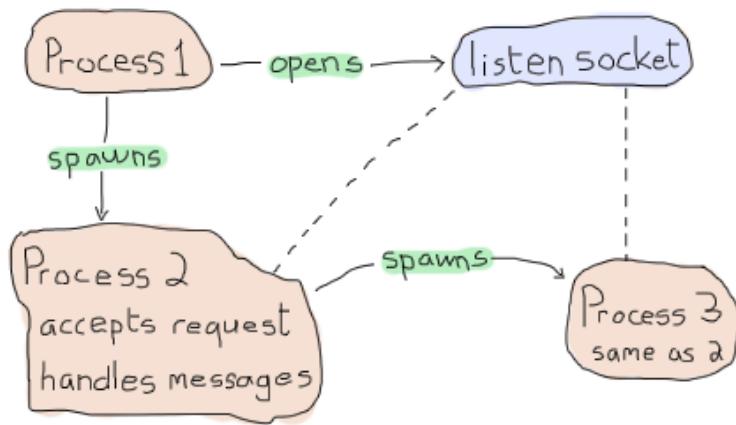
%% Echoing back whatever was obtained
handle(Socket) ->
    inet:setopts(Socket, [{active, once}]),
```

```

receive
{tcp, Socket, <<"quit", _/binary>>} ->
    gen_tcp:close(Socket);
{tcp, Socket, Msg} ->
    gen_tcp:send(Socket, Msg),
    handle(Socket)
end.

```

To understand how this works, a little graphical representation might be helpful:



So the `start_server` function opens a listen socket, spawns an acceptor and then just idles forever. The idling is necessary because the listen socket is bound to the process that opened it, so that one needs to remain alive as long as we want to handle connections. Each acceptor process waits for a connection to accept. Once one connection comes in, the acceptor process starts a new similar process and shares the listen socket to it. Then it can move on and do some processing while the new guy's working. Each handler will repeat all messages it gets until one of them starts with "quit" — then the connection is closed.

Note: the pattern `<<"quit", _/binary>>` means that we first want to match on a binary string containing the characters q, u, i, and t, plus some binary data we don't care about `(_)`.

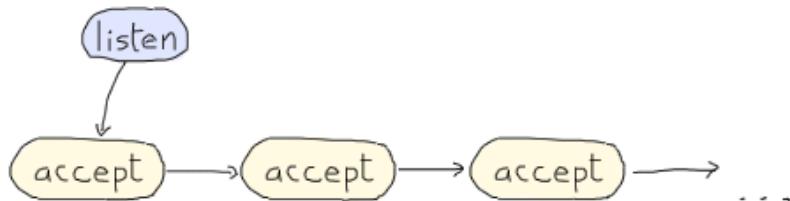
Start the server in an Erlang shell by doing `naive_tcp:start_server(8091)`. Then open up a telnet client (remember, telnet clients are technically not for raw TCP, but act as good clients to test servers without having to write one) to localhost and you can see the following taking place:

```
$ telnet localhost 8091
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^>'.
hey there
hey there
that's what I asked
that's what I asked
stop repeating >:( 
stop repeating >:( 
quit doing that!
Connection closed by foreign host.
```

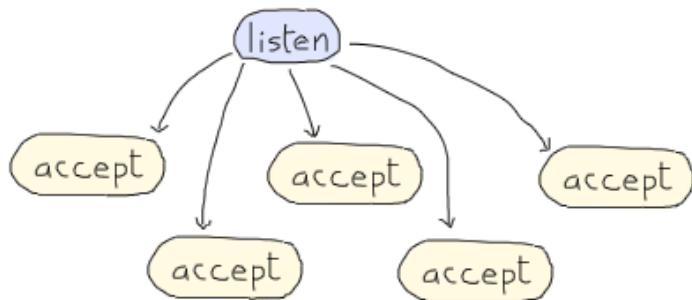
Hooray. Time to start a new company called *Poople Inc.* and launch a few social networks with such a server. Except that as the name of the module mentions it, this is a naive implementation. The code is simple, but wasn't thought with parallelism in mind. If all the requests come one by one, then the naive server works fine. What happens if we have a queue of 15 people wanting to connect to the server at once, though?

Then only one query at a time can be replied to, and this has to do with each process first waiting for the connection, setting it up, then spawning a new acceptor. The 15th request in the queue will have had to wait for 14 other connections to have been set up to even get the chance of asking for a right to discuss with our server. If you're working with production servers, it might be closer to, I don't know, five hundred to a thousand queries per second. That's impractical.

What we'd need would be to change the sequential workflow we have:



To something more parallel:



By having many acceptors already ready on standby, we'll be cutting down on a lot of delays to answer new queries. Now, rather than going through another demo implementation, we'll study sockserv-1.0.1 from the last chapter. It will be nicer to explore something based on real OTP components and real world practice. In fact, the general pattern of sockserv is the same one used in servers like cowboy (although cowboy is no doubt more reliable than sockserv) and the utorrent torrent client.

To build this Process Quest's sockserv, we'll go top-down. The scheme we'll need will have to be a supervisor with many workers. If we look at the parallel drawing above, the supervisor should hold the listen socket and share it to all workers, which will be in charge of accepting things.

How do we write a supervisor that can share things across all workers? There is no way to do it with regular supervision: all children are entirely independent, no matter if you use `one_for_one`, `one_for_all` or

`rest_for_one` supervision. A natural reflex could be to turn to some global state: a registered process that just holds the listen socket and hands it over to the handlers. You must fight this reflex and be clever. Use the force (and the ability to read back into the [supervisors chapter](#)). You've got 2 minutes to think of a solution (the timing of the two minutes is based on the honor system. Time it yourself.)

The secret is in using a `simple_one_for_one` supervisor. Because the `simple_one_for_one` supervisors share the child specification with all of its children, all we need to do is shove the listen socket in there for all the children to access it!

So here's the supervisor in all its glory:

```
%%% The supervisor in charge of all the socket acceptors.  
-module(sockserv_sup).  
-behaviour(supervisor).  
  
-export([start_link/0, start_socket/0]).  
-export([init/1]).  
  
start_link() ->  
    supervisor:start_link({local, ?MODULE}, ?MODULE, []).  
  
init([]) ->  
    {ok, Port} = application:get_env(port),  
    %% Set the socket into {active_once} mode.  
    %% See sockserv_serv comments for more details  
    {ok, ListenSocket} = gen_tcp:listen(Port, [{active,once}, {packet,line}]),  
    spawn_link(fun empty_listeners/0),  
    {ok, {{simple_one_for_one, 60, 3600},  
        [{socket,  
            {sockserv_serv, start_link, [ListenSocket]}, % pass the socket!  
            temporary, 1000, worker, [sockserv_serv]}  
       ]}}.  
  
start_socket() ->
```

```
supervisor:start_child(?MODULE, []).  
  
%% Start with 20 listeners so that many multiple connections can  
%% be started at once, without serialization. In best circumstances,  
%% a process would keep the count active at all times to insure nothing  
%% bad happens over time when processes get killed too much.  
empty_listeners() ->  
    [start_socket() || _ <- lists:seq(1,20)],  
    ok.
```

So what is going on in here. The standard `start_link/0` and `init/1` functions are there. You can see `sockserv` getting the `simple_one_for_one` restart strategy, and the child specification having `ListenSocket` passed around. Every child started with `start_socket/0` will have it as an argument by default. Magic!

Just having that won't be enough. We want the application to be able to serve queries as soon as possible. That's why I added that call to `spawn_link(fun empty_listeners/0)`. The `empty_listeners/0` function will start 20 handlers to be locked and waiting for incoming connections. I've put it inside a `spawn_link/1` call for a simple reason: the supervisor process is in its `init/1` phase and cannot answer any messages. If we were to call ourselves from within the `init` function, the process would deadlock and never finish running. An external process is needed just for this reason.

Note: In the snippet above, you'll notice I pass the option `{packet, line}` to `gen_tcp`. This option will make it so all received packets will be broken into separate lines and queued up based on that (the line ends will still be part of the received strings). This will help make sure things work better with telnet clients in our case. Be aware, however, that lines longer than the receive buffer may be split over many packets, so it is possible for two packets to represent a single line. Verifying that the received content ends in a newline will let you know if the line is over or not.

So yeah, that was the whole tricky part. We can now focus on writing the workers themselves.

If you recall the Process Quest sessions from last chapter, things went this way:

1. The user connects to the server
2. The server asks for the character's name
3. The user sends in a character name
4. The server suggests stats
5.
 1. the user refuses, go back to point 4
 2. the user accepts, go to point 6
6. The game sends event to the player, until:
7. The user sends quit to the server or the socket is forced close

This means we will have two kinds of input to our server processes: input coming from the Process Quest application and input coming from the user. Data coming from the user will be doing so from a socket and so will be handled in our gen_server's handle_info/2 function. Data coming from Process Quest can be sent in a way we control, and so a cast handled by handle_cast will make sense there. First, we must start the server:

```
-module(sockserv_serv).  
-behaviour(gen_server).  
  
-record(state, {name, % player's name  
               next, % next step, used when initializing  
               socket}). % the current socket  
  
-export([start_link/1]).  
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,  
       code_change/3, terminate/2]).
```

First of all is a pretty standard gen_server callback module. The only special thing here is the state containing the character's name, the

socket, and a field called `next`. The `next` part is a bit of a catch-all field to store temporary information related to the state of the server. A `gen_fsm` could have possibly been used here without too much trouble.

For the actual server startup:

```
-define(TIME, 800).  
-define(EXP, 50).
```

```
start_link(Socket) ->  
    gen_server:start_link(?MODULE, Socket, []).  
  
init(Socket) ->  
    %% properly seeding the process  
    <<A:32, B:32, C:32>> = crypto:rand_bytes(12),  
    random:seed({A,B,C}),  
    %% Because accepting a connection is a blocking function call,  
    %% we can not do it in here. Forward to the server loop!  
    gen_server:cast(self(), accept),  
    {ok, #state{socket=Socket}}.  
  
%% We never need you, handle_call!  
handle_call(_E, _From, State) ->  
    {noreply, State}.
```

The two macros defined above (`?TIME` and `?EXP`) are special parameters that make it possible to set the baseline delay between actions (800 milliseconds) and the amount of experience required to reach the second level (50, doubled after each level).

You'll notice that the `start_link/1` function takes a socket. That's the listen socket passed in from `sockserv_sup`.

The first bit about the random seed is about making sure a process is properly seeded to later generate character statistics. Otherwise, some default value will be used across many processes and we don't

want that. The reason why we're initializing in the `init/1` function rather than in whatever library that uses random numbers is because seeds are stored at a process-level (damn it! mutable state!) and we wouldn't want to set a new seed on each library call.

In any case, the real important bit there is that we're casting a message to ourselves. The reason for this is that `gen_tcp:accept/1-2` is a blocking operation, combined with the fact that all `init` functions are synchronous. If we wait 30 seconds to accept a connection, the supervisor starting the process will also be locked 30 seconds. So yeah, we cast a message to ourselves, then add the listen socket to the state's `socket` field.

Don't Drink Too Much Kool-Aid:

If you read code from other people, you will often see people calling `random:seed/1` with the result of `now()`. `now()` is a nice function because it returns monotonic time (always increasing, never twice the same). However, it's a bad seed value for the random algorithm used in Erlang. For this reason, it's better to use `crypto:rand_bytes(12)` to generate 12 crypto-safe random bytes (use `crypto:strong_rand_bytes(12)` if you're on R14B03+). By doing `<<A:32, B:32, C:32>>`, we're casting the 12 bytes to 3 integers to be passed in.

Update:

Starting with version 18.0, the `rand` module has been introduced, containing better pseudo-random algorithms than the `random` module, and without needing to be seeded.

We need to accept that connection. Enough fooling around:

```
handle_cast({accept, S = #state{socket=ListenSocket}}) ->
    {ok, AcceptSocket} = gen_tcp:accept(ListenSocket),
    %% Remember that thou art dust, and to dust thou shalt return.
    %% We want to always keep a given number of children in this app.
    sockserv_sup:start_socket(), % a new acceptor is born, praise the lord
```

```
send(AcceptSocket, "What's your character's name?", []),
{noreply, S#state{socket=AcceptSocket, next=name}};
```

We accept the connection, start a replacement acceptor (so that we always have about 20 acceptors ready to handle new connections), then store the accept socket as a replacement to *ListenSocket* and note that the next message we receive through a socket is about a name with the 'next' field.

But before moving on, we send a question to the client through the `send` function, defined as follows:

```
send(Socket, Str, Args) ->
    ok = gen_tcp:send(Socket, io_lib:format(Str++"~n", Args)),
    ok = inet:setopts(Socket, [{active, once}]),
    ok.
```

Trickery! Because I expect us to pretty much always have to reply after receiving a message, I do the *active once* routine within that function, and also add line breaks in there. Just laziness locked in a function.

We've completed steps 1 and 2, and now we have to wait for user input coming from the socket:

```
handle_info({tcp, _Socket, Str}, S = #state{next=name}) ->
    Name = line(Str),
    gen_server:cast(self(), roll_stats),
    {noreply, S#state{name=name, next=stats}};
```

We have no idea what's going to be in the *Str* string, but that's alright because the `next` field of the state lets us know whatever we receive is a name. Because I was expecting users to use telnet for the demo application, all bits of text we're going to receive will contain line ends. The `line/1` function, defined as follows, strips them away:

```
%% Let's get rid of the white space and ignore whatever's after.
```

```
%% makes it simpler to deal with telnet.
```

```
line(Str) ->
```

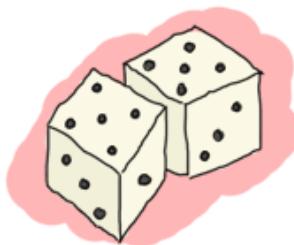
```
    hd(string:tokens(Str, "\r\n")).
```

Once we've received that name, we store it and then cast a message to ourselves (`roll_stats`) to generate stats for the player, the next step in line.

Note: if you look in the file, you'll see that instead of matching on entire messages, I've used a shorter `?SOCK(var)` macro. The macro is defined as `-define(SOCK(Msg), {tcp, _Port, Msg}).` and is just a quick way for someone as lazy as I am to match on strings with slightly less typing.

The stats rolling comes back into a `handle_cast` clause:

```
handle_cast(roll_stats, S = #state{socket=Socket}) ->
    Roll = pq_stats:initial_roll(),
    send(Socket,
        "Stats for your character:~n"
        " Charisma: ~B~n"
        " Constitution: ~B~n"
        " Dexterity: ~B~n"
        " Intelligence: ~B~n"
        " Strength: ~B~n"
        " Wisdom: ~B~n~n"
        "Do you agree to these? y/n~n",
        [Points || {_Name, Points} <- lists:sort(Roll)],
        {noreply, S#state{next={stats, Roll}}};
```



The pq_stats module contains functions to roll stats, and the whole clause is only being used to output the stats there. The ~B format parameters means we want an integer to be printed out. The next part of the state is a bit overloaded here. Because we ask the user whether they agree or not, we will have to wait for them to tell us so, and either drop the stats and generate new ones, or pass them to the Process Quest character we'll no doubt start very soon.

Let's listen to the user input, this time in the handle_info function:

```
handle_info({tcp, Socket, Str}, S = #state{socket=Socket, next={stats, _}}) ->
    case line(Str) of
        "y" ->
            gen_server:cast(self(), stats_accepted);
        "n" ->
            gen_server:cast(self(), roll_stats);
        _ -> % ask again because we didn't get what we wanted
            send(Socket, "Answer with y (yes) or n (no)", [])
    end,
    {noreply, S};
```

It would have been tempting to start the character in this direct function clause, but I decided against it: handle_info is to handle user input, handle_cast for Process Quest things. Separation of concerns! If the user denies the stats, we just call roll_stats again. Nothing new. When the user accepts, then we can start the Process Quest character and start waiting for events from there:

```
%% The player has accepted the stats! Start the game!
handle_cast(stats_accepted, S = #state{name>Name, next={stats, Stats}}) ->
    processquest:start_player(Name, [{stats, Stats}, {time, ?TIME},
                                      {lvlexp, ?EXP}]),
    processquest:subscribe(Name, sockserv_pq_events, self()),
    {noreply, S#state{next=playing}};
```

Those are regular calls I defined for the game. You start a player, and subscribe to the events with the sockserv_pq_events event handler.

The next state is playing, which means that all messages received are more than likely to be from the game:

```
%% Events coming in from process quest
%% We know this because all these events' tuples start with the
%% name of the player as part of the internal protocol defined for us
handle_cast(Event, S = #state{name=N, socket=Sock}) when element(1, Event) == N ->
    [case E of
        {wait, Time} -> timer:sleep(Time);
        IoList -> send(Sock, IoList, [])
    end || E <- sockserv_trans:to_str(Event)], % translate to a string
    {noreply, S}.
```

I won't get into the details of how this works too much. Just know that `sockserv_trans:to_str(Event)` convert some game event to lists of IO lists or `{wait, Time}` tuples that represent delays to wait between parts of events (we print executing a ... messages a bit before showing what the item dropped by the enemy is).

If you recall the list of steps to follow, we've covered them all except one. Quitting when a user tells us they want to. Put the following clause as the top one in `handle_info`:

```
handle_info({tcp, _Socket, "quit"++_}, S) ->
    processquest:stop_player(S#state.name),
    gen_tcp:close(S#state.socket),
    {stop, normal, S};
```

Stop the character, close the socket, terminate the process. Hooray. Other reasons to quit include the TCP socket being closed by the client:

```
handle_info({tcp_closed, _Socket}, S) ->
    {stop, normal, S};
handle_info({tcp_error, _Socket, _}, S) ->
    {stop, normal, S};
handle_info(E, S) ->
```

```
io:format("unexpected: ~p~n", [E]),  
{noreply, S}.
```

I also added an extra clause to handle unknown messages. If the user types in something we don't expect, we don't want to crash. Only the `terminate/2` and `code_change/3` functions are left to do:

```
code_change(_OldVsn, State, _Extra) ->  
{ok, State}.

terminate(normal, _State) ->  
ok;  
terminate(_Reason, _State) ->  
io:format("terminate reason: ~p~n", [_Reason]).
```

If you followed through the whole thing, you can try compiling this file and substituting it for the corresponding beam file in the release we had and see if it runs well. It should, if you copied things right (and if I did too).

Where to go From Now?

Your next assignment, if you are to accept it, is to add a few more commands of your choice to the client: why not add things like 'pause' that will queue up actions for a while and then output them all once you resume the server? Or if you're bad ass enough, noting the levels and stats you have so far in the `sockserv_serv` module, and adding commands to fetch them from the client side. I always hated exercises left to the reader, but sometimes it's just too tempting to drop one here and there, so enjoy!

Otherwise, reading the source of existing server implementations, programming some yourself and whatnot will all be good exercises. Rare are the languages where doing things like writing a web server is an exercise for amateurs, but Erlang is one of them. Practice a bit and it'll become like a second nature. Erlang communicating to the

outside world is just one of the many steps we've done towards writing useful software.

EUnited Nations Council

The Need for Tests



The software we've written has gotten progressively bigger and somewhat more complex with time. When that happens, it becomes rather tedious to start an Erlang shell, type things in, look at results, and make sure things work after code has been changed. As time goes on, it becomes simpler for everyone to run tests that are all prepared and ready in advance rather than following checklists of stuff to check by hand all the time. These are usually pretty good reasons to want tests in your software. It's also possible that you're a fan of test-driven development and so will also find tests useful.

If you recall the chapter where we wrote a [RPN calculator](#), we had a few tests that we had manually written. They were simply a set of pattern matches of the form `Result = Expression` that would crash if something went wrong, or would succeed otherwise. That works for simple bits of code you write for yourself, but when we get to more serious tests, we will definitely want something better, like a framework.

For unit tests, we'll tend to stick to *EUnit* (which we see in this chapter). For integration tests, EUnit as well as *Common Test* can both do the job. In fact, Common Test can do everything from unit tests up to system tests, and even testing of external software, not written in Erlang. For now we'll go with EUnit, given how simple it is for the good results it yields.

EUnit, What's a EUnit?

EUnit, in its simplest form, is just a way to automate running functions that end in `_test()` in a module by assuming they are unit tests. If you go dig out that RPN calculator I mentioned above, you'll find the following code:

```
rpn_test() ->
  5 = rpn("2 3 +"),
  87 = rpn("90 3 -"),
  -4 = rpn("10 4 3 + 2 * -"),
  -2.0 = rpn("10 4 3 + 2 * - 2 /"),
  ok = try
    rpn("90 34 12 33 55 66 + * - +")
  catch
    error:{badmatch,[_|_]} -> ok
  end,
  4037 = rpn("90 34 12 33 55 66 + * - + -"),
  8.0 = rpn("2 3 ^"),
  true = math:sqrt(2) == rpn("2 0.5 ^"),
  true = math:log(2.7) == rpn("2.7 ln"),
  true = math:log10(2.7) == rpn("2.7 log10"),
  50 = rpn("10 10 10 20 sum"),
  10.0 = rpn("10 10 10 20 sum 5 /"),
  1000.0 = rpn("10 10 20 0.5 prod"),
  ok.
```

This was the test function we wrote to make sure the calculator worked fine. Find the old module and try this:

```
1> c(calc).  
{ok,calc}  
2> eunit:test(calc).  
  Test passed.  
ok
```

Calling `eunit:test(Module)`. was all we needed! Yay, we now know EUnit! Pop the champagne and let's head to a different chapter!

Obviously a testing framework that only does this little wouldn't be very useful, and in technical programmer jargon, it might be described as 'not very good'. EUnit does more than automatically exporting and running functions ending in `_test()`. For one, you can move the tests out to a different module so that your code and its tests are not mixed together. This means you can't test private functions anymore, but also means that if you develop all your tests against the module's interface (the exported functions), then you won't need to rewrite tests when you refactor your code. Let's try separating tests and code with two simple modules:

```
-module(ops).  
-export([add/2]).  
  
add(A,B) -> A + B.  
  
-module(ops_tests).  
-include_lib("eunit/include/eunit.hrl").
```

```
add_test() ->
  4 = ops:add(2,2).
```

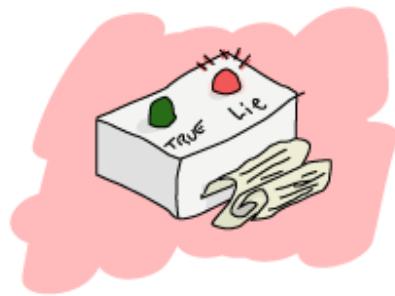
So we have ops and ops_tests, where the second includes tests related to the first. Here's a thing EUnit can do:

```
3> c(ops).
{ok,ops}
4> c(ops_tests).
{ok,ops_tests}
5> eunit:test(ops).
  Test passed.
ok
```

Calling `eunit:test(Mod)` automatically looks for `Mod_tests` and runs the tests within that one. Let's change the test a bit (make it `3 = ops:add(2,2)`) to see what failures look like:

```
6> c(ops_tests).
{ok,ops_tests}
7> eunit:test(ops).
ops_tests: add_test (module 'ops_tests')...*failed*
:error:{badmatch,4}
  in function ops_tests:add_test/0
```

```
=====
Failed: 1. Skipped: 0. Passed: 0.
error
```



We can see what test failed (`ops_tests: add_test...`) and why it did (`::error:{badmatch,4}`). We also get a full report of how many tests passed or failed. The output is pretty bad though. At least as bad as regular Erlang crashes: no line numbers, no clear explanation (4 didn't match with what, exactly?), etc. We're left helpless by a test framework that runs tests but doesn't tell you much about them.

For this reason, EUnit introduces a few macros to help us. Each of them will give us cleaner reporting (including line numbers) and clearer semantics. They're the difference between knowing that something goes wrong and knowing *why* something goes wrong:

`?assert(Expression)`, `?assertNot(Expression)`

Will test for boolean values. If any value other than `true` makes it into `?assert`, an error will be shown. Same for `?assertNot`, but for negative values. This macro is somewhat equivalent to `true = X or false = Y`.

`?assertEqual(A, B)`

Does a strict comparison (equivalent to `=:=`) between two expressions, `A` and `B`. If they are different, a failure will occur. This is roughly equivalent to `true = X =:= Y`. Since

R14B04, the macro `?assertNotEqual` is available to do the opposite of `?assertEqual`.

`?assertMatch(Pattern, Expression)`

This allows us to match in a form similar to `Pattern = Expression`, without variables ever binding. This means that I could do something like `?assertMatch({X,X}, some_function())` and assert that I receive a tuple with two elements being identical. Moreover, I could later do `?assertMatch(X,Y)` and X would not be bound.

This is to say that rather than properly being like `Pattern = Expression`, what we have is closer to `(fun (Pattern) -> true; (_) -> erlang:error(nomatch) end)(Expression)`: variables in the pattern's head *never* get bound across multiple assertions. The macro `?assertNotMatch` has been added to EUnit in R14B04.

`?assertError(Pattern, Expression)`

Tells EUnit that *Expression* should result in an error. As an example, `?assertError(badarith, 1/0)` would be a successful test.

`?assertThrow(Pattern, Expression)`

Exactly the same as `?assertError`, but with `throw(Pattern)` instead of `erlang:error(Pattern)`.

`?assertExit(Pattern, Expression)`

Exactly the same as `?assertError`, but with `exit(Pattern)` (and not `exit/2`) instead of `erlang:error(Pattern)`.

`?assertException(Class, Pattern, Expression)`

A general form of the three previous macros. As an example, `?assertException(error, Pattern, Expression)` is the same as `?assertError(Pattern, Expression)`. Starting with R14B04, there is also the macro `?assertNotException/3` available for tests.

Using these macros, we could write better tests in our module:

```

-module(ops_tests).
-include_lib("eunit/include/eunit.hrl").

add_test() ->
    4 = ops:add(2,2).

new_add_test() ->
    ?assertEqual(4, ops:add(2,2)),
    ?assertEqual(3, ops:add(1,2)),
    ?assert(is_number(ops:add(1,2))),
    ?assertEqual(3, ops:add(1,1)),
    ?assertError(badarith, 1/0).

```

And running them:

```

8> c(ops_tests).
./ops_tests.erl:12: Warning: this expression will fail with a 'badarith' exception
{ok,ops_tests}
9> eunit:test(ops).
ops_tests: new_add_test...*failed*
::error:{assertEqual_failed,[{module,ops_tests},
                           {line,11},
                           {expression,"ops : add ( 1, 1 )"},
                           {expected,3},
                           {value,2}]}
in function ops_tests:'-new_add_test/0-fun-3-'1
in call from ops_tests:new_add_test/0

```

```

=====
Failed: 1. Skipped: 0. Passed: 1.
error
```

See how much nicer the error reporting is? We know that the `assertEqual` on line 11 of `ops_tests` failed. When we called `ops:add(1,1)`,

we thought we'd receive 3 as a value, but we instead got 2. Of course you've got to read these values as Erlang terms, but at least they're there.

What's annoying with this, however, is that even though we had 5 assertions, only one failed but the whole test was still considered a failure. It would be nicer to know that some assertion failed without behaving as if all the others after it failed too. Our test is the equivalent of taking an exam in school, and as soon as you make a mistake, you fail and get thrown out of school. Then your dog dies and you just have a horrible day.

Test Generators

Because of this common need for flexibility, EUnit supports something called *test generators*. Test generators are pretty much shorthand for assertions wrapped in functions that can be run later, in clever manners. Instead of having functions ending with `_test()` with macros that are of the form `?assertSomething`, we will use functions that end in `_test_()` and macros of the form `?_assertSomething`. Those are small changes, but they make things much more powerful. The two following tests would be equivalent:

```
function_test() -> ?assert(A == B).  
function_test_() -> ?_assert(A == B).
```

Here, `function_test_()` is called a *test generator function*, while `?_assert(A == B)` is called a *test generator*. It is called that way, because secretly, the underlying implementation of `?_assert(A`

`== B) is fun() -> ?assert(A == B) end.` That is to say, a function that generates a test.

The advantage of test generators, compared to regular assertions, is that they are funs. This means that they can be manipulated without being executed. We could, in fact, have *test sets* of the following form:

```
my_test_() ->
  [?_assert(A),
   ?_assert(B),
   ?_assert(C),
   [?_assert(D)]],
  [[?_assert(E)]].
```

Test sets can be deeply nested lists of test generators. We could have functions that return tests! Let's add the following to `ops_tests`:

```
add_test_() ->
  [test_them_types(),
   test_them_values(),
   ?_assertError(badarith, 1/0)].  
  
test_them_types() ->
  ?_assert(is_number(ops:add(1,2))).  
  
test_them_values() ->
  [?_assertEqual(4, ops:add(2,2)),
   ?_assertEqual(3, ops:add(1,2)),
   ?_assertEqual(3, ops:add(1,1))].
```

Because only `add_test_()` ends in `_test_()`, the two functions `test_them_Something()` will not be seen as tests. In fact, they will

only be called by `add_test_()` to generate tests:

```
1> c(ops_tests).
./ops_tests.erl:12: Warning: this expression will fail with a 'badarith' exception
./ops_tests.erl:17: Warning: this expression will fail with a 'badarith' exception
{ok,ops_tests}
2> eunit:test(ops).
ops_tests:25: test_them_values...*failed*
[...]
ops_tests: new_add_test...*failed*
[...]
```

```
=====
Failed: 2. Skipped: 0. Passed: 5.
```

error

So we still get the expected failures, and now you see that we jumped from 2 tests to 7. The magic of test generators.

What if we only wanted to test some parts of the suite, maybe just `add_test_/0`? Well EUnit has a few tricks up its sleeves:

```
3> eunit:test({generator, fun ops_tests:add_test_/0}).
ops_tests:25: test_them_values...*failed*
:error:{assertEqual_failed,[{module,ops_tests},
                           {line,25},
                           {expression,"ops : add ( 1,1 )"},{expected,3},
                           {value,2}]}}

in function ops_tests:'-test_them_values/0-fun-4-/1
```

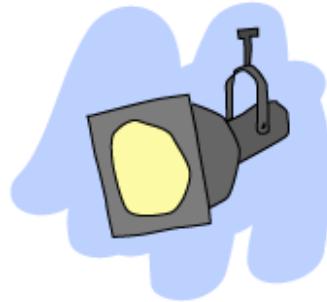
```
=====
Failed: 1. Skipped: 0. Passed: 4.
```

error

Note that this only works with test generator functions. What we have here as `{generator, Fun}` is what EUnit parlance calls a *test representation*. We have a few other representations:

- `{module, Mod}` runs all tests in `Mod`
- `{dir, Path}` runs all the tests for the modules found in `Path`
- `{file, Path}` runs all the tests found in a single compiled module
- `{generator, Fun}` runs a single generator function as a test, as seen above
- `{application, AppName}` runs all the tests for all the modules mentioned in `AppName`'s `.app` file.

These different test representations can make it easy to run test suites for entire applications or even releases.



Fixtures

It would still be pretty hard to test entire applications just by using assertions and test generators. This is why *fixtures* were added. Fixtures, while not being a catch-all solution to getting your tests up and running to the application level, allow you to build a certain scaffolding around tests.

The scaffolding in question is a general structure that allows us to define setup and teardown functions for each of the test. These functions will allow you to build the state and environment required for each of the tests to be useful. Moreover, the scaffolding will let you specify how to run the tests (do you want to run them locally, in separate processes, etc.?)

There are a few types of fixtures available, with variations to them. The first type is simply called the *setup* fixture. A setup fixture takes one of the many following forms:

```
{setup, Setup, Instantiator}  
{setup, Setup, Cleanup, Instantiator}  
{setup, Where, Setup, Instantiator}  
{setup, Where, Setup, Cleanup, Instantiator}
```

Argh! It appears we need a little bit of EUnit vocabulary in order to understand this (this will be useful if you need to go read the EUnit documentation):

Setup

A function that takes no argument. Each of the tests will be passed the value returned by the setup function.

Cleanup

A function that takes the result of a setup function as an argument, and takes care of cleaning up whatever is needed. If in OTP terminate does the opposite of init, then cleanup functions are the opposite of setup functions for EUnit.

Instantiator

It's a function that takes the result of a setup function and returns a test set (remember, test sets are possibly deeply nested lists of ?_Macro assertions).

Where

Specifies how to run the tests: local, spawn, {spawn, node()}.

Alright, so what does this look like in practice? Well, let's imagine some test to make sure that a fictive process registry correctly handles trying to register the same process twice, with different names:

```
double_register_test_() ->
    {setup,
     fun start/0,          % setup function
     fun stop/1,           % teardown function
     fun two_names_one_pid/1}. % instantiator

start() ->
    {ok, Pid} = registry:start_link(),
    Pid.

stop(Pid) ->
    registry:stop(Pid).

two_names_one_pid(Pid) ->
    ok = registry:register(Pid, quite_a_unique_name, self()),
    Res = registry:register(Pid, my_other_name_is_more_creative, self()),
    [?_assertEqual({error, already_named}, Res)].
```

This fixture first starts the registry server within the start/0 function. Then, the instantiator

two_names_one_pid(ResultFromSetup) is called. In that test, the only thing I do is try to register the current process twice.

That's where the instantiator does its work. The result of the second registration is stored in the variable `Res`. The function will then return a test set containing a single test (`?_assertEqual({error, already_named}, Res)`). That test set will be run by EUnit. Then, the teardown function `stop/1` will be called. Using the pid returned by the setup function, it'll be able to shut down the registry that we had started beforehand. Glorious!

What's even better is that this whole fixture itself can be put inside a test set:

```
some_test_() ->
  [{setup, fun start/0, fun stop/1, fun some_instantiator1/1},
   {setup, fun start/0, fun stop/1, fun some_instantiator2/1},
   ...
   {setup, fun start/0, fun stop/1, fun some_instantiatorN/1}].
```

And this will work! What's annoying there is the need to always repeat that setup and teardown functions, especially when they're always the same. That's where the second type of fixture, the *foreach* fixture, enters the stage:

```
{foreach, Where, Setup, Cleanup, [Instantiator]}
{foreach, Setup, Cleanup, [Instantiator]}
{foreach, Where, Setup, [Instantiator]}
{foreach, Setup, [Instantiator]}
```

The *foreach* fixture is quite similar to the *setup* fixture, with the difference that it takes lists of instantiators. Here's the `some_test_/0` function written with a *foreach* fixture:

```
some2_test_() ->
  {foreach,
```

```
fun start/0,  
fun stop/1,  
[fun some_instantiator1/1,  
 fun some_instantiator2/1,  
 ...  
 fun some_instantiatorN/1]}.
```

That's better. The foreach fixture will then take each of the instantiators and run the setup and teardown function for each of them.

Now we know how to have a fixture for one instantiator, then a fixture for many of them (each getting their setup and teardown function calls). What if I want to have one setup function call, and one teardown function calls for many instantiators?

In other words, what if I have many instantiators, but I want to set some state only once? There's no easy way for this, but here's a little trick that might do it:

```
some_tricky_test_() ->  
{setup,  
 fun start/0,  
 fun stop/1,  
 fun (SetupData) ->  
 [some_instantiator1(SetupData),  
 some_instantiator2(SetupData),  
 ...  
 some_instantiatorN(SetupData)]  
end}.
```

By using the fact that test sets can be deeply nested lists, we wrap a bunch of instantiators with an anonymous function

behaving like an instantiator for them.



Tests can also have some finer grained control into how they should be running when you use fixtures. Four options are available:

{spawn, TestSet}

Runs tests in a separate process than the main test process. The test process will wait for all of the spawned tests to finish

{timeout, Seconds, TestSet}

The tests will run for *Seconds* number of Seconds. If they take longer than *Seconds* to finish, they will be terminated without further ado.

{inorder, TestSet}

This tells EUnit to run the tests within the test set strictly in the order they are returned.

{inparallel, Tests}

Where possible, the tests will be run in parallel.

As an example, the `some_tricky_test_/0` test generator could be rewritten as follows:

```
some_tricky2_test_() ->
    {setup,
     fun start/0,
     fun stop/1,
     fun(SetupData) ->
        {inparallel,
         [some_instantiator1(SetupData),
          some_instantiator2(SetupData),
          ...
          some_instantiatorN(SetupData)]}
    end}.
```

That's really most of it for fixtures, but there's one more nice trick I've forgot to show for now. You can give descriptions of tests in a neat way. Check this out:

```
double_register_test_() ->
    {"Verifies that the registry doesn't allow a single process to "
     "be registered under two names. We assume that each pid has the "
     "exclusive right to only one name",
     {setup,
      fun start/0,
      fun stop/1,
      fun two_names_one_pid/1}}.
```

Nice, huh? You can wrap a fixture by doing `{Comment, Fixture}` in order to get readable tests. Let's put this in practice.

Testing Regis

Because just seeing fake tests as above isn't the most entertaining thing to do, and because pretending to test software that doesn't exist is even worse, we'll instead study the tests I have written for the regis-1.0.0 process registry, the one used by Process Quest.



Now, the development of regis was done in a test-driven manner. Hopefully you don't hate TDD (Test-Driven Development), but even if you do, it shouldn't be too bad because we'll look at the test suite after the fact. By doing this, we cut through the few trial-and-error sequences and backpedaling that I might have had writing it the first time and I'll look like I'm really competent, thanks to the magic of text editing.

The regis application is made of three processes: a supervisor, a main server, and then an application callback module. Knowing that the supervisor will only check the server and that the application callback module will do nothing except behaving as an interface for the two other modules, we can

safely write a test suite focusing on the server itself, without any external dependencies.

Being a good TDD fan, I begun by writing a list of all the features I wanted to cover:

- Respect an interface similar to the Erlang default process registry
- The Server will have a registered name so that it can be contacted without tracking its pid
- A process can be registered through our service and can then be contacted by its name
- A list of all registered processes can be obtained
- A name that is not registered by any process should return the atom 'undefined' (much like the regular Erlang registry) in order to crash calls using them
- A process can not have two names
- Two processes can not share the same name
- A process that was registered can be registered again if it was unregistered between calls
- Unregistering a process never crashes
- A registered process' crash will unregister its name

That's a respectable list. Doing the elements one by one and adding cases as I went, I transformed each of the specification into a test. The final file obtained was `regis_server_tests`. I wrote things using a basic structure a bit like this:

```
-module(regis_server_tests).  
-include_lib("eunit/include/eunit.hrl").
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%  
%%% TESTS DESCRIPTIONS %%%  
%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%  
%%% SETUP FUNCTIONS %%%  
%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%  
%%% ACTUAL TESTS %%%  
%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%  
%%% HELPER FUNCTIONS %%%  
%%%%%%%%%%%%%%%%%%%%%%%%%
```

Ok, I give it to you, that looks weird when the module is empty, but as you fill it up, it makes more and more sense.

After adding a first test, the initial one being that it should be possible to start a server and access it by name, the file looked like this:

```
-module(regis_server_tests).  
-include_lib("eunit/include/eunit.hrl").
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%  
%%% TESTS DESCRIPTIONS %%%  
%%%%%%%%%%%%%%%%%%%%%%%%%  
start_stop_test_() ->  
    {"The server can be started, stopped and has a registered name",  
     {setup,  
      fun start/0,  
      fun stop/1,  
      fun is_registered/1}}.
```

```

%%%%%%%%%%%%%%%
%%% SETUP FUNCTIONS %%%
%%%%%%%%%%%%%%%
start() ->
    {ok, Pid} = regis_server:start_link(),
    Pid.

stop(_) ->
    regis_server:stop().

%%%%%%%%%%%%%%%
%%% ACTUAL TESTS %%%
%%%%%%%%%%%%%%%
is_registered(Pid) ->
    [?_assert(erlang:is_process_alive(Pid)),
     ?_assertEqual(Pid, whereis(regis_server))].


%%%%%%%%%%%%%%%
%%% HELPER FUNCTIONS %%%
%%%%%%%%%%%%%%%

```

See the organization now? Already so much better. The top part of the file contains only fixtures and top-level description of features. The second part contains setup and cleanup functions that we might need. The last one contains the instantiators returning test sets.

In this case, the instantiator checks to see whether `regis_server:start_link()` spawned a process that was truly alive, and that it was registered with the name `regis_server`. If it's true, then that will work for the server.

If we look at the current version of the file, it now looks more like this for the two first sections:

```
-module(regis_server_tests).
-include_lib("eunit/include/eunit.hrl").

-define(setup(F), {setup, fun start/0, fun stop/1, F}).

%%%%%%%%%%%%%%%
%%% TESTS DESCRIPTIONS %%%
%%%%%%%%%%%%%%%

start_stop_test_() ->
    {"The server can be started, stopped and has a registered name",
     ?setup(fun is_registered/1)}.

register_test_() ->
    [{"A process can be registered and contacted",
      ?setup(fun register_contact/1)},
     {"A list of registered processes can be obtained",
      ?setup(fun registered_list/1)},
     {"An undefined name should return 'undefined' to crash calls",
      ?setup(fun noregister/1)},
     {"A process can not have two names",
      ?setup(fun two_names_one_pid/1)},
     {"Two processes cannot share the same name",
      ?setup(fun two_pids_one_name/1)}].

unregister_test_() ->
    [{"A process that was registered can be registered again iff it was "
      "unregistered between both calls",
      ?setup(fun re_un_register/1)},
     {"Unregistering never crashes",
      ?setup(fun unregister_nocrash/1)},
     {"A crash unregisters a process",
```

```

?setup(fun crash_unregister/1}]}.

%%%%%%%%%%%%%%%
%%% SETUP FUNCTIONS %%%
%%%%%%%%%%%%%%%
start() ->
    {ok, Pid} = regis_server:start_link(),
    Pid.

stop(_) ->
    regis_server:stop().

%%%%%%%%%%%%%%%
%%% HELPER FUNCTIONS %%%
%%%%%%%%%%%%%%%
%% nothing here yet

```

Nice, isn't it? Note that as I was writing the suite, I ended up seeing that I never needed any other setup and teardown functions than `start/0` and `stop/1`. For this reason, I added the `?setup(Instantiator)` macro, that makes things look a bit better than if all the fixtures were to be fully expanded. It's now pretty obvious that I turned each point of the feature list into a bunch of tests. You'll note that I divided all tests depending on whether they had to do with starting and stopping the server (`start_stop_test_/0`), registering processes (`register_test_/0`) and unregistering processes (`unregister_test_/0`).

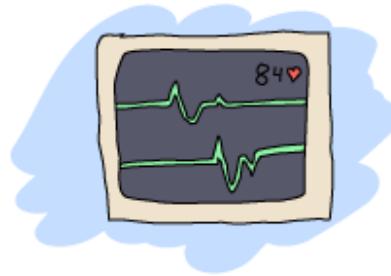
By reading the test generators' definitions, we can know what the module is supposed to be doing. The tests become documentation (although they should not replace proper documentation).

We'll study the tests a bit and see why things were done in a certain way. The first test in the list `start_stop_test_/0`, with the simple requirement that the server can be registered:

```
start_stop_test_() ->
    {"The server can be started, stopped and has a registered name",
     ?setup(fun is_registered/1)}.
```

The implementation of the test itself is put in the `is_registered/1` function:

```
%%%%%
%%% ACTUAL TESTS %%%
%%%%%
is_registered(Pid) ->
    [?_assert(erlang:is_process_alive(Pid)),
     ?_assertEqual(Pid, whereis(regis_server))].
```



As explained earlier when we looked at the first version of the test, this checks whether the process is available or not. There's nothing really special about that one, although the function `erlang:is_process_alive(Pid)` might be new to you. As its name says, it checks whether a process is currently running. I've put that test in there for the simple reason that it might well be possible that the server crashes as soon as we start it, or that it's never started in the first place. We don't want that.

The second test is related to being able to register a process:

```
{"A process can be registered and contacted",
?setup(fun register_contact/1)}
```

Here's what the test looks like:

```
register_contact(_) ->
  Pid = spawn_link(fun() -> callback(regcontact) end),
  timer:sleep(15),
  Ref = make_ref(),
  WherePid = regis_server:whereis(regcontact),
  regis_server:whereis(regcontact) ! {self(), Ref, hi},
  Rec = receive
    {Ref, hi} -> true
    after 2000 -> false
  end,
  [?_assertEqual(Pid, WherePid),
  ?_assert(Rec)].
```

Granted, this isn't the most elegant test around. What it does is that it spawns a process that will do nothing but register itself and reply to some message we send it. This is all done in the callback/1 helper function defined as follows:

```
%%%%%%%%%%%%%
%%% HELPER FUNCTIONS %%%
%%%%%%%%%%%%%
callback(Name) ->
  ok = regis_server:register(Name, self()),
  receive
    {From, Ref, Msg} -> From ! {Ref, Msg}
  end.
```

So the function has the module register itself, receives a message, and sends a response back. Once the process is started, the `register_contact/1` instantiator waits 15 milliseconds (just a tiny delay to make sure the other process registers itself), and then tries to use the `whereis` function from `regis_server` to retrieve a Pid and send a message to the process. If the regis server is functioning correctly, a message will be received back and the pids will match in the tests at the bottom of the function.

Don't Drink Too Much Kool-Aid:

By reading that test, you have seen the little timer work we've had to do. Because of the concurrent and time-sensitive nature of Erlang programs, tests will frequently be filled with tiny timers like that that have the sole role of trying to synchronise bits of code.

The problem then becomes to try and define what should be considered a good timer, a delay that is long enough. With a system running many tests or even a computer under heavy load, will the timers still be waiting for long enough?

Erlang programmers who write tests sometimes have to be clever in order to minimize how much synchronisation they need to get things to work. There is no easy solution for it.

The next tests are introduced as follows:

```
{"A list of registered processes can be obtained",
?setup(fun registered_list/1)}
```

So when a bunch of processes have been registered, it should be possible to get a list of all the names. This is a functionality similar to Erlang's `registered()` function call:

```
registered_list(_) ->
    L1 = regis_server:get_names(),
    Pids = [spawn(fun() -> callback(N) end) || N <- lists:seq(1,15)],
    timer:sleep(200),
    L2 = regis_server:get_names(),
    [exit(Pid, kill) || Pid <- Pids],
    [?_assertEqual([], L1),
     ?_assertEqual(lists:sort(lists:seq(1,15)), lists:sort(L2))].
```

First of all, we make sure that the first list of registered processes is empty (`?_assertEqual(L1, [])`) so that we've got something that works even when no process has ever tried to register itself. Then 15 processes are created, all of which will try to register themselves with a number (1..15). We make the test sleep a bit to make sure all processes have the time to register themselves, and then call `regis_server:get_names()`. The names should include all integers between 1 and 15, inclusively. Then a slight cleanup is done by eliminating all the registered processes — we don't want to be leaking them, after all.



You'll notice the tendency of the tests to store state in variables (L_1 and L_2) before using them in test sets. The reason for this is that the test set that is returned is executed well after the test initiator (the whole active bit of code) has been running. If you were to try and put function calls that depend on other processes and time-sensitive events in the `?_assert*` macros, you'd get everything out of sync and things would generally be awful for you and the people using your software.

The next test is simple:

```
{"An undefined name should return 'undefined' to crash calls",
?setup(fun noregister/1)}
```

...

```
noregister(_) ->
  [_assertError(badarg, regis_server:whereis(make_ref()) ! hi),
   _assertEqual(undefined, regis_server:whereis(make_ref()))].
```

As you can see, this tests for two things: we return `undefined`, and the specification's assumption that using `undefined` does indeed crash attempted calls. For that one, there is no need to use temporary variables to store the state: both tests can be executed at any time during the life of the `regis` server given we never change its state.

Let's keep going:

```
{"A process can not have two names",
?setup(fun two_names_one_pid/1)},
```

...

```
two_names_one_pid(_) ->
    ok = regis_server:register(make_ref(), self()),
    Res = regis_server:register(make_ref(), self()),
    [?_assertEqual({error, already_named}, Res)].
```

That's pretty much the same test we used in a demo in the previous section of the chapter. In this one, we're just looking to see whether we get the right output and that the test process can't register itself twice with different names.

Note: you might have noticed that the tests above tend to use `make_ref()` a whole lot. When possible, it is useful to use functions that generate unique values like `make_ref()` does. If at some point in the future someone wants to run tests in parallel or to run them under a single regis server that never stops, then it will be possible to do so without needing to modify the tests.

If we were to use hard coded names like `a`, `b`, and `c` in all the tests, then it would be very likely that sooner or later, name conflicts would happen if we were to try and run many test suites at once. Not all tests in the `regis_server_tests` suite follow this advice, mostly for demonstration purposes.

The next tests is the opposite of `two_names_one_pid`:

```
{"Two processes cannot share the same name",
?setup(fun two_pids_one_name/1)}].
```

...

```
two_pids_one_name(_) ->
```

```
Pid = spawn(fun() -> callback(myname) end),
timer:sleep(15),
Res = regis_server:register(myname, self()),
exit(Pid, kill),
[_assertEqual({error, name_taken}, Res)].
```

Here, because we need two processes and the results of only one of them, the trick is to spawn one process (the one whose results we do not need), and then do the critical part ourselves.

You can see that timers are used to make sure that the other process tries registering a name first (within the `callback/1` function), and that the test process itself waits to try at its turn, expecting an error tuple (`{error, name_taken}`) as a result.

This covers all the features for the tests related to the registration of processes. Only those related to unregistering processes are left:

```
unregister_test_() ->
[{"A process that was registered can be registered again iff it was "
 "unregistered between both calls",
 ?setup(fun re_un_register/1)},
 {"Unregistering never crashes",
 ?setup(fun unregister_nocrash/1)},
 {"A crash unregisters a process",
 ?setup(fun crash_unregister/1)}].
```

Let's see how they are to be implemented. The first one is kind of simple:

```
re_un_register(_) ->
Ref = make_ref(),
L = [regis_server:register(Ref, self()),
```

```
regis_server:register(make_ref(), self()),  
regis_server:unregister(Ref),  
regis_server:register(make_ref(), self())],  
[?_assertEqual([ok, {error, already_named}], ok, ok], L)].
```

This way of serializing all the calls in a list is a nifty trick I like to do when I need to test the results of all the events. By putting them in a list, I can then compare the sequence of actions to the expected [ok, {error, already_named}, ok, ok] to see how things went. Note that there is nothing specifying that Erlang should evaluate the list in order, but the trick above has pretty much always worked.

The following test, the one about never crashing, goes like this:

```
 unregister_nocrash(_) ->  
 ?_assertEqual(ok, regis_server:unregister(make_ref())).
```

Whoa, slow down here, buddy! That's it? Yes it is. If you look back at `re_un_register`, you'll see that it already handles testing the 'unregistration' of processes. For `unregister_nocrash`, we really only want to know if it will work to try and remove a process that's not there.

Then comes the last test, and one of the most important ones for any test registry you'll ever have: a named process that crashes will have the name unregistered. This has serious implications, because if you didn't remove names, you'd end up having an ever growing registry server with an ever shrinking name selection:

```
crash_unregister(_)->  
 Ref = make_ref(),
```

```
Pid = spawn(fun() -> callback(Ref) end),
timer:sleep(150),
Pid = regis_server:whereis(Ref),
exit(Pid, kill),
timer:sleep(95),
regis_server:register(Ref, self()),
S = regis_server:whereis(Ref),
Self = self(),
?_assertEqual(Self, S).
```

This one reads sequentially:

1. Register a process
2. Make sure the process is registered
3. Kill that process
4. Steal the process' identity (the true spy way)
5. Check whether we do hold the name ourselves.

In all honesty, the test could have been written in a simpler manner:

```
crash_unregister(_) ->
Ref = make_ref(),
Pid = spawn(fun() -> callback(Ref) end),
timer:sleep(150),
Pid = regis_server:whereis(Ref),
exit(Pid, kill),
?_assertEqual(undefined, regis_server:whereis(Ref)).
```

That whole part about stealing the identity of the dead process was nothing but a petty thief's fantasy.

That's it! If you've done things right, you should be able to compile the code and run the test suite:

```
$ erl -make
Recompile: src/regis_sup
...
$ erl -pa ebin/
1> eunit:test(regis_server).
All 13 tests passed.
ok
2> eunit:test(regis_server, [verbose]).
===== EUnit =====
module 'regis_server'
module 'regis_server_tests'
  The server can be started, stopped and has a registered name
    regis_server_tests:49: is_registered...ok
    regis_server_tests:50: is_registered...ok
    [done in 0.006 s]
...
[done in 0.520 s]
=====
All 13 tests passed.
ok
```

Oh yeah, see how adding the 'verbose' option will add test descriptions and run time information to the reports? That's neat.



He Who Knits EUnits

In this chapter, we've seen how to use most features of EUnit, how to run suites written in them. More importantly, we've seen a few techniques related to how to write tests for concurrent processes, using patterns that make sense in the real world.

One last trick should be known: when you feel like testing processes such as gen_servers and gen_fsms, you might feel like inspecting the internal state of the processes. Here's a nice trick, courtesy of the sys module:

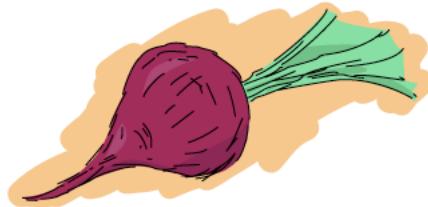
```
3> regis_server:start_link().
{ok,<0.160.0>}
4> regis_server:register(shell, self()).
ok
5> sys:get_status(whereis(regis_server)).
{status,<0.160.0>,
 {module,gen_server},
 [{['$ancestors',[<0.31.0>]}],
  {'$initial_call',[{regis_server,init,1}]}],
  running,<0.31.0>:[],
  [{header,"Status for generic server regis_server"},
   {data,[{"Status",running},
         {"Parent",<0.31.0>},
         {"Logged events",[]}]}],
  {data,[{"State",
         {state,{1,{<0.31.0>,{shell,#Ref<0.0.0.333>},nil,nil}}},
         {1,{shell,{<0.31.0>,#Ref<0.0.0.333>},nil,nil}}]}]}]
```

Neat, huh? Everything that has to do with the server's innards is given to you: you can now inspect everything you need, all the time!

If you feel like getting more comfortable with testing servers and whatnot, I recommend reading the tests written for Process Quests' player module. They test the gen_server using a different technique, where all individual calls to `handle_call`, `handle_cast` and `handle_info` are tried independently. It was still developed in a test-driven manner, but the needs of that one forced things to be done differently.

In any case, we'll see the true value of tests when we rewrite the process registry to use ETS, an in-memory database available for all Erlang processes.

Bears, ETS, Beets

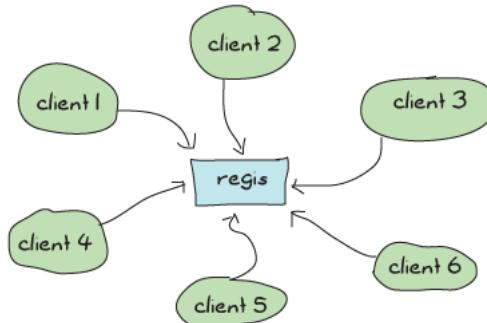


Something we've been doing time and time again has been to implement some kind of storage device as a process. We've done fridges to store things, built regis to register processes, seen key/value stores, etc. If we were programmers doing object-oriented design, we would be having a bunch of singletons floating around, and special storage classes and whatnot. In fact, wrapping data structures like dicts and gb_trees in processes is a bit like that.

Holding data structures in a process is actually fine for a lot of cases — whenever we actually need that data to do some task within the process, as internal state, and so on. We've had plenty of valid uses and we shouldn't change that. However, there is one case where it is possibly not the best choice: when the process holds a data structure for the sake of sharing it with other processes and little more.

One of the applications we've written is guilty of that. Can you guess which? Of course you can; I've mentioned it at the end of last chapter: regis needs to be rewritten. It's not that it doesn't work or can't do its job well, but because it acts as a gateway to share data with potentially *a lot* of other processes, there is an architectural problem with it.

See, regis is this central application to do messaging in Process Quest (and anything else that would use it), and pretty much every message going to a named process has to go through it. This means that even though we took great care to make our applications very concurrent with independent actors and made sure our supervision structure was right to scale up, all of our operations will depend on a central regis process that will need to answer messages one by one:



If we have a lot of message passing going on, regis risks getting busier and busier, and if the demand is high enough our whole system will become sequential and slow. That's pretty bad.

Note: we have no direct proof that regis is a bottleneck within Process Quest — In fact, Process Quest does very little messaging compared to many other applications in the wild. If we were using regis for something that requires a lot more messaging and lookups, then the problems would be more apparent.

The few ways we'd have to get around that would be to either split regis into subprocesses to make lookups faster by sharding the data, or find a way to store the data in some database that will allow for parallel and concurrent access of the data. While the first way to do it would be very interesting to explore, we'll go through an easier path by doing the latter.

Erlang has something called ETS (Erlang Term Storage) tables. ETS tables are an efficient in-memory database included with the Erlang virtual machine. It sits in a part of the virtual machine where destructive updates are allowed and where garbage collection dares not approach. They're generally fast, and a pretty easy way for Erlang programmers to optimize some of their code when parts of it get too slow.

ETS tables allow limited concurrency in reads and writes (much better than none at all for a process' mailbox) in a way that could let us optimize away a lot of the pain.

Don't Drink Too Much Kool-Aid:

While ETS tables are a nice way to optimize, they should still be used with some care. By default, the VM is limited to 1400 ETS tables. While it is possible to change that number (`erl -env ERL_MAX_ETS_TABLES Number`), this default low level is a good sign that you should try to avoid having one table per process in general

But before we rewrite regis to use ETS, we should try to understand a bit of ETS' principles beforehand.

The Concepts of ETS

ETS tables are implemented as BIFs in the `ets` module. The main design objectives ETS had was to provide a way to store large amounts of data in Erlang with constant access time (functional data structures usually tend to flirt with logarithmic access time) and to have such storage look as if it were implemented as processes in order to keep their use simple and idiomatic.

Note: having tables look like processes doesn't mean that you can spawn them or link to them, but that they can respect semantics of nothing-shared, wrapping calls behind functional interfaces, having them handle any native data type for Erlang, and having the possibility to give them names (in a separate registry), etc.

All ETS tables natively store Erlang tuples containing whatever you want, where one of the tuple elements will act as a primary key that you use to sort things. That is to say, having tuples of people of the form {Name, Age, PhoneNumber, Email} will let you have a table that looks like:

```
{Name, Age, PhoneNumber, Email},  
...
```

So if we say that we want to have the table's index be the e-mail addresses, we can do this by telling ETS to set the key position to 4 (we'll see how to do this in a bit, when we get to actual ETS function calls). Once you've decided on a key, you can choose different ways to store data into tables:

set

A set table will tell you that each key instance must be unique. There can be no duplicate e-mail in the database above. Sets are great when you need to use a standard key/value store with constant time access.

ordered_set

There can still only be one key instance per table, but ordered_set adds a few other interesting properties. The first is that elements in an ordered_set table will be ordered (who would have thought?!). The first element of the table is the smallest one, and the last element is the largest one. If you traverse a table iteratively (jumping to the next element over and over again), the values should be increasing, which is not necessarily true of set tables. Ordered set tables are great when you frequently need to operate on ranges (I want entries 12 to 50!). They will, however, have the downside of being slower in their access time ($O(\log N)$ where N is the number of objects stored).

bag

A bag table can have multiple entries with the same key, as long as the tuples themselves are different. This means that the table can have {key, some, values} and {key, other, values} inside of it without a problem, which would be impossible with sets (they have the same key). However, you couldn't have {key, some, values} twice in the table as they would be entirely identical.

duplicate_bag

The tables of this type work like bag tables, with the exception that they do allow entirely identical tuples to be held multiple time within the same table.

Note: ordered_set tables will see the values 1 and 1.0 as identical for all operations. Other tables will see them as different.

The last general concept to learn about is that ETS tables will have the concept of a controlling process, much like sockets do. When a process calls a function that starts a new

ETS table, that process is the owner of the table.

By default, only the owner of the table can write to it, but everyone can read from it. This is known as the *protected* level of permissions. You can also choose to set the permissions to *public*, where everyone can read and write, or *private*, where only the owner can read or write.



The concept of table ownership goes a bit further. The ETS table is intimately linked to the process. If the process dies, the table disappears (and so does all of its content). However, the table can be given away, much like we did with sockets and their controlling processes, or a heir can be determined so that if the owner process dies, the table is automatically given away to the heir process.

ETS Phone Home

To start an ETS table, the function `ets:new/2` has to be called. The function takes the argument `Name` and then a list of options. In return, what you get is a unique identifier necessary to use the table, comparable to a `Pid` for processes. The options can be any of these:

Type = set | ordered_set | bag | duplicate_bag

Sets the type of table you want to have, as described in the previous section. The default value is `set`.

Access = private | protected | public

Lets us set the permissions on the table as described earlier. The default option is `protected`.

named_table

Funnily enough, if you call `ets:new(some_name, [])`, you'll be starting a protected set table, without a name. For the name to be used as a way to contact a table (and to be made unique), the option `named_table` has to be passed to the function. Otherwise, the name of the table will purely be for documentation purposes and will appear in functions such as `ets:i()`, which print information about all ETS tables in the system.

{keypos, Position}

As you may (and should) recall, ETS tables work by storing tuples. The *Position* parameter holds an integer from 1 to *N* telling which of each tuple's element shall act as the primary key of the database table. The default key position is set to 1. This means you have to be careful if you're using records as each record's first element is always going to be the record's name (remember what they look like in their tuple form). If you want to use any field as the key, use {keypos, #RecordName.FieldName}, as it will return the position of *FieldName* within the record's tuple representation.

{heir, Pid, Data} | {heir, none}

As mentioned in the previous section, ETS tables have a process that acts as their parent. If the process dies, the table disappears. If the data attached to a table is something you might want to keep alive, then defining a heir can be useful. If the process attached to a table dies, the heir receives a message saying {ETS-TRANSFER, TableId, FromPid, Data}, where *Data* is the element passed when the option was first defined. The table is automatically inherited by the heir. By default, no heir is defined. It is possible to define or change a heir at a later point in time by calling ets:setopts(Table, {heir, Pid, Data}) or ets:setopts(Table, {heir, none}). If you simply want to give the table away, call ets:give_away/3.

{readConcurrency, true | false}

This is an option to optimize the table for read concurrency. Setting this option to true means that reads become way cheaper to do, but then make switching to writes a lot more expensive. Basically, this option should be enabled when you do a lot of reading and little writing and need an extra kick of performance. If you do some reading, some writing and they are interleaved, using this option might even hurt performance.

{writeConcurrency, true | false}

Usually, writing to a table will lock the whole thing and nobody else can access it, either for reading or writing to it, until the write is done. Setting this option to 'true' lets both reads and writes be done concurrently, without affecting the ACID properties of ETS. Doing this, however, will reduce the performance of sequential writes by a single process and also the capacity of concurrent reads. You can combine this option with 'readConcurrency' when both writes and reads come in large bursts.

compressed

Using this option will allow the data in the table to be compressed for most fields, but not the primary key. This comes at the cost of performance when it comes to inspecting entire elements of the table, as we will see with the next functions.

Then, the opposite of table creation is table destruction. For that one, all that's needed is to call ets:delete(Table) where *Table* is either a table id or the name of a named table. If you want to delete a single entry from the table, a very similar function call is required: ets:delete(Table, Key).

Two more functions are required for very basic table handling: insert(Table, ObjectOrObjects) and lookup(Table, Key). In the case of insert/2, *ObjectOrObjects* can be either a single tuple or a list of tuples to insert:

```

1> ets:new(ingredients, [set, named_table]).  

ingredients  

2> ets:insert(ingredients, {bacon, great}).  

true  

3> ets:lookup(ingredients, bacon).  

[{:bacon,great}]  

4> ets:insert(ingredients, [{bacon, awesome}, {cabbage, alright}]).  

true  

5> ets:lookup(ingredients, bacon).  

[{:bacon,awesome}]  

6> ets:lookup(ingredients, cabbage).  

[{:cabbage,alright}]  

7> ets:delete(ingredients, cabbage).  

true  

8> ets:lookup(ingredients, cabbage).  

[]

```

You'll notice that the `lookup` function returns a list. It will do that for all types of tables, even though set-based tables will always return at most one item. It just means that you should be able to use the `lookup` function in a generic way even when you use bags or duplicate bags (which may return many values for a single key).

Another thing that takes place in the snippet above is that inserting the same key twice overwrites it. This will always happen in sets and ordered sets, but not in bags or duplicate bags. If you want to avoid this, the function `ets:insert_new/2` might be what you want, as it will only insert elements if they are not in the table already.

Note: The tuples do not have to all be of the same size in an ETS table, although it should be seen as good practice to do so. It is however necessary that the tuple is at least of the same size (or greater) than whatever the key position is.

There's another `lookup` function available if you need to only fetch part of a tuple. The function is `lookup_element(TableID, Key, PositionToReturn)` and it will either return the element that matched (or a list of them if there is more than one with a bag or duplicate bag table). If the element isn't there, the function errors out with `badarg` as a reason.

In any case let's try again with a bag:

```

9> TabId = ets:new(ingredients, [bag]).  

16401  

10> ets:insert(TabId, {bacon, delicious}).  

true  

11> ets:insert(TabId, {bacon, fat}).  

true  

12> ets:insert(TabId, {bacon, fat}).  

true  

13> ets:lookup(TabId, bacon).  

[{:bacon,delicious},{:bacon,fat}]

```

As this is a bag, {bacon, fat} is only there once even though we inserted twice, but you can see that we can still have more than one 'bacon' entry. The other thing to look at here is that without passing in the `named_table` option, we have to use the `TableId` to use the table.

Note: if at any point while copying these examples your shell crashes, the tables are going to disappear as their parent process (the shell) has disappeared.

The last basic operations we can make use of will be about traversing tables one by one. If you're paying attention, `ordered_set` tables are the best fit for this:

```
14> ets:new(ingredients, [ordered_set, named_table]).  
ingredients  
15> ets:insert(ingredients, [{ketchup, "not much"}, {mustard, "a lot"}, {cheese, "yes", "goat"}, {patty, "moose"}, {onions, "c  
true  
16> Res1 = ets:first(ingredients).  
cheese  
17> Res2 = ets:next(ingredients, Res1).  
ketchup  
18> Res3 = ets:next(ingredients, Res2).  
mustard  
19> ets:last(ingredients).  
patty  
20> ets:prev(ingredients, ets:last(ingredients)).  
onions
```

As you can see, elements are now in sorting order, and they can be accessed one after the other, both forwards and backwards. Oh yeah, and then we need to see what happens in boundary conditions:

```
21> ets:next(ingredients, ets:last(ingredients)).  
'$end_of_table'  
22> ets:prev(ingredients, ets:first(ingredients)).  
'$end_of_table'
```

When you see atoms starting with a \$, you should know that they're some special value (chosen by convention) by the OTP team telling you about something. Whenever you're trying to iterate outside of the table, you'll see these `$end_of_table` atoms.

So we know how to use ETS as a very basic key-value store. There are more advanced uses now, when we need more than just matching on keys.

Meeting Your Match



There are plenty of functions to be used with ETS when it comes to finding records from more special mechanisms.

When we think about it, the best way to select things would be with pattern matching, right? The ideal scenario would be to be able to somehow store a pattern to match on within a variable (or as a data structure), pass that to some ETS function and let the said function do its thing.

This is called *higher order pattern matching* and sadly, it is not available in Erlang. In fact, very few languages have it. Instead, Erlang has some kind of sublanguage that Erlang programmers have agreed to that is being used to describe pattern matching as a bunch of regular data structures.

This notation is based on tuples to fit nicely with ETS. It simply lets you specify variables (regular and "don't care" variables), that can be mixed with the tuples to do pattern matching. Variables are written as '\$0', '\$1', '\$2', and so on (the number has no importance except in how you'll get the results) for regular variables. The "don't care" variable can be written as '_'. All these atoms can take form in a tuple like:

```
{items, '$3', '$1', '_', '$3'}
```

This is roughly equivalent to saying {items, C, A, _, C} with regular pattern matching. As such, you can guess that the first element needs to be the atom items, that the second and fifth slots of the tuple need to be identical, etc.

To make use of this notation in a more practical setting, two functions are available: match/2 and match_object/2 (there are match/3 and match_object/3 available as well, but their use is outside the scope of this chapter and readers are encouraged to check the docs for details.) The former will return the variables of the pattern, while the later will return the whole entry that matched the pattern.

```
1> ets:new(table, [named_table, bag]).  
table  
2> ets:insert(table, [{items, a, b, c, d}, {items, a, b, c, a}, {cat, brown, soft, loveable, selfish}, {friends, [jenn,jeff,etc]}], {item  
true  
3> ets:match(table, {items, '$1', '$2', '_', '$1'}).  
[[a,b],[1,2]]  
4> ets:match(table, {items, '$114', '$212', '_', '$6'}).  
[[d,a,b],[a,a,b],[1,1;2]]  
5> ets:match_object(table, {items, '$1', '$2', '_', '$1'}).  
[{items,a,b,c,a},{items,1,2,3,1}]  
6> ets:delete(table).  
true
```

The nice thing about match/2-3 as a function is that it only returns what is strictly necessary to be returned. This is useful because as mentioned earlier, ETS tables are following the nothing-shared ideals. If you have very large records, only copying the necessary fields might be a good thing to do. Anyway, you'll also notice that while the numbers in variables

have no explicit meaning, their order is important. In the final list of values returned, the value bound to `$14` will always come after the values bound to `$6` by the pattern. If nothing matches, empty lists are returned.

It is also possible you might want to delete entries based on such a pattern match. In these cases, the function `ets:match_delete(Table, Pattern)` is what you want.



This is all fine and lets us put any kind of value to do basic pattern matching in a weird way. It would be pretty neat if it were possible to have things like comparisons and ranges, explicit ways to format the output (maybe lists isn't what we want), and so on. Oh wait, you can!

You Have Been Selected

This is when we get something more equivalent to true function heads-level pattern matching, including very simple guards. If you've ever used a SQL database before, you might have seen ways to do queries where you compare elements that are greater, equal, smaller, etc. than other elements. This is the kind of good stuff we want here.

The people behind Erlang thus took the syntax we've seen for matches and augmented it in crazy ways until it was powerful enough. Sadly, they also made it unreadable. Here's what it can look like:

```
[{{'$1','$2',<<1>>,'$3','$4},  
 [{"andalso',{'>', '$4,150}, {'<', '$4,500}],  
 {"orelse,{=='$2,meat}, {=='$2,dairy}}]],  
 ['$1']],  
 [{{$1,$2',<<1>>,'$3','$4},  
 [{"<', '$3,4.0},{is_float,$3}]],  
 ['$1']]}
```

This is pretty ugly, not the data structure you would want your children to look like. Believe it or not, we'll learn how to write these things called *match specifications*. Not under that form, no, that would be a bit too hard for no reason. We'll still learn how to read them though! Here's what it looks like a bit from a higher level view:

```
[{InitialPattern1, Guards1, ReturnedValue1},  
 {InitialPattern2, Guards2, ReturnedValue2}].
```

Or from a yet higher view:

```
[clause1,  
 clause2]
```

So yeah, things like that represent, roughly, the pattern in a function head, then the guards, then the body of a function. The format is still limited to '\$N' variables for the initial pattern, exactly the same to what it was for match functions. The new sections are the guard patterns, allowing to do something quite similar to regular guards. If we look closely to the guard `[{'<', '$3', 4.0}, {is_float, '$3'}]`, we can see that it is quite similar to ... when `Var < 4.0, is_float(Var) -> ...` as a guard.

The next guard, more complex this time, is:

```
[{andalso,[{>,$4,150},{<,$4,500}],  
 {orelse,[{==,$2,meat},{==,$2,dairy}]}
```

Translating it gives us a guard that looks like ... when `Var4 > 150 andalso Var4 < 500, Var2 == meat orelse Var2 == dairy -> Got it?`

Each operator or guard function works with a prefix syntax, meaning that we use the order `{FunctionOrOperator, Arg1, ..., ArgN}`. So `is_list(X)` becomes `{is_list, '$1'}`, `X andalso Y` becomes `{andalso, X, Y}`, and so on. Reserved keywords such as `andalso`, `orelse` and operators like `==` need to be put into atoms so the Erlang parser won't choke on them.

The last section of the pattern is what you want to return. Just put the variables you need in there. If you want to return the full input of the match specification, use the variable `'$'_` to do so. A full specification of match specifications can be found in the Erlang Documentation.

As I said before, we won't learn how to write patterns that way, there's something nicer to do it. ETS comes with what is called a *parse transform*. Parse transforms are an undocumented (thus not supported by the OTP team) way of accessing the Erlang parse tree halfway through the compiling phase. They let ballsy Erlang programmers transform the code in a module to a new alternative form. Parse transforms can be pretty much anything and change existing Erlang code to almost anything else, as long as it doesn't change the language's syntax or its tokens.

The parse transform coming with ETS needs to be enabled manually for each module that needs it. The way to do it in a module is as follows:

```
-module(SomeModule).  
-include_lib("stdlib/include/ms_transform.hrl").  
...  
some_function() ->  
    ets:fun2ms(fun(X) when X > 4 -> X end).
```

The line `-include_lib("stdlib/include/ms_transform.hrl")`. contains some special code that will override the meaning of `ets:fun2ms(SomeLiteralFun)` whenever it's being used in a module. Rather than

being a higher order function, the parse transform will analyse what is in the fun (the pattern, the guards and the return value), remove the function call to `ets:fun2ms/1`, and replace it all with an actual match specification. Weird, huh? The best is that because this happens at compile time, there is no overhead to using this way of doing things.

We can try it in the shell, without the include file this time:

```
1> ets:fun2ms(fun(X) -> X end).
[{$1,[],[$1]}]
2> ets:fun2ms(fun({X,Y}) -> X+Y end).
[{$1,'$2},[],[{$1,'$1','$2}]]]
3> ets:fun2ms(fun({X,Y}) when X < Y -> X+Y end).
[{$1,'$2},[{$1,'$2},[{$1,'$1','$2}]]]
4> ets:fun2ms(fun({X,Y}) when X < Y, X rem 2 == 0 -> X+Y end).
[{$1,'$2},
 [{$1,'$1','$2},{$1,'$1,'$2},0],
 [{$1,'$1,'$2}]]]
5> ets:fun2ms(fun({X,Y}) when X < Y, X rem 2 == 0; Y == 0 -> X end).
[{$1,'$2},
 [{$1,'$1,$2},{$1,'$1,'$2},0],
 [{$1}],{$1,'$2},[{$1,'$2,'$2},0],[{$1}]]]
```

All of these! They are written so easily now! And of course the funs are much simpler to read. How about that complex example from the beginning of the section? Here's what it would be like as a fun:

```
6> ets:fun2ms(fun({Food, Type, <<1>>, Price, Calories}) when Calories > 150 andalso Calories < 500, Type == meat orelse
[{$1,'$2,<<1>>,$3,$4},
 [{andalso,{>,$4,150},{<,$4,500}},
 {orelse,{==,$2,meat},{==,$2,dairy}}],
 [{$1}],
 [{$1,'$2,<<1>>,$3,$4},
 [{<,$3,4.0},{is_float,$3}],
 [{$1}]]]
```

It doesn't exactly make sense at first glance, but at least it's much simpler to figure out what it means when variables can actually have a name rather than a number. One thing to be careful about is that not all funs are valid match specifications:

```
7> ets:fun2ms(fun(X) -> my_own_function(X) end).
Error: fun containing the local function call 'my_own_function/1' (called in body) cannot be translated into match_specification
{error,transform_error}
8> ets:fun2ms(fun(X,Y) -> ok end).
Error: ets:fun2ms requires fun with single variable or tuple parameter
{error,transform_error}
9> ets:fun2ms(fun([X,Y]) -> ok end).
Error: ets:fun2ms requires fun with single variable or tuple parameter
{error,transform_error}
10> ets:fun2ms(fun({<<X/binary>>}) -> ok end).
```

```
Error: fun head contains bit syntax matching of variable 'X', which cannot be translated into match_spec  
{error,transform_error}
```

The function head needs to match on a single variable or a tuple, no non-guard functions can be called as part of the return value, assigning values from within binaries is not allowed, etc. Try stuff in the shell, see what you can do.

Don't Drink Too Much Kool-Aid:

A function like `ets:fun2ms` sounds totally awesome, right! You have to be careful with it. A problem with it is that if `ets:fun2ms` can handle dynamic funs when in the shell (you can pass funs around and it will just eat them up), this isn't possible in compiled modules.

This is due to the fact that Erlang has two kinds of funs: shell funs and module funs. Module funs are compiled down to some compact format understood by the virtual machine. They're opaque and cannot be inspected to know how they are on the inside.

On the other hand, shell funs are abstract terms not yet evaluated. They're made in a way that the shell can call the evaluator on them. The function `fun2ms` will thus have two versions of itself: one for when you're getting compiled code, and one from when you're in the shell.

This is fine, except that the funs aren't interchangeable with different types of funs. This means that you can't take a compiled fun and try to call `ets:fun2ms` on it while in the shell, and you can't take a dynamic fun and send it over to a compiled bit of code that's calling `fun2ms` in there. Too bad!

To make match specifications useful, it would make sense to use them. This can be done by using the functions `ets:select/2` to fetch results, `ets:select_reverse/2` to get results in reverse in ordered_set tables (for other types, it's the same as `select/2`), `ets:select_count/2` to know how many results match the specification, and `ets:select_delete(Table, MatchSpec)` to delete records matching a match specification.

Let's try it, first defining a record for our tables, and then populating them with various goods:

```
11> rd(food, {name, calories, price, group}).  
food  
12> ets:new(food, [ordered_set, {keypos,#food.name}, named_table]).  
food  
13> ets:insert(food, [#food{name=salmon, calories=88, price=4.00, group=meat},  
13> #food{name=cereals, calories=178, price=2.79, group=bread},  
13> #food{name=milk, calories=150, price=3.23, group=dairy},  
13> #food{name=cake, calories=650, price=7.21, group=delicious},  
13> #food{name=bacon, calories=800, price=6.32, group=meat},  
13> #food{name=sandwich, calories=550, price=5.78, group=whatever}]).  
true
```

We can then try to select food items under a given number of calories:

```
14> ets:select(food, ets:fun2ms(fun(N = #food{calories=C}) when C < 600 -> N end)).  
[#food{name = cereals,calories = 178,price = 2.79,group = bread},
```

```
#food{name = milk,calories = 150,price = 3.23,group = dairy},
#food{name = salmon,calories = 88,price = 4.0,group = meat},
#food{name = sandwich,calories = 550,price = 5.78,group = whatever}]
15> ets:select_reverse(food, ets:fun2ms(fun(N = #food{calories=C}) when C < 600 -> N end)).
[#food{name = sandwich,calories = 550,price = 5.78,group = whatever},
#food{name = salmon,calories = 88,price = 4.0,group = meat},
#food{name = milk,calories = 150,price = 3.23,group = dairy},
#food{name = cereals,calories = 178,price = 2.79,group = bread}]
```

Or maybe what we want is just delicious food:

```
16> ets:select(food, ets:fun2ms(fun(N = #food{group=delicious}) -> N end)).
[#food{name = cake,calories = 650,price = 7.21,group = delicious}]
```

Deleting has a little special twist to it. You have to return `true` in the pattern instead of any kind of value:

```
17> ets:select_delete(food, ets:fun2ms(fun(#food{price=P}) when P > 5 -> true end)).
```

3

```
18> ets:select_reverse(food, ets:fun2ms(fun(N = #food{calories=C}) when C < 600 -> N end)).
[#food{name = salmon,calories = 88,price = 4.0,group = meat},
#food{name = milk,calories = 150,price = 3.23,group = dairy},
#food{name = cereals,calories = 178,price = 2.79,group = bread}]
```

And as the last selection shows, items over \$5.00 were removed from the table.

There are way more functions inside ETS, such as ways to convert the table to lists or files (`ets:tab2list/1`, `ets:tab2file/1`, `ets:file2tab/1`), get information about all tables (`ets:i/0`, `ets:info(Table)`). Heading over to the official documentation is strongly recommended in this case.

There's also an application called `observer` that has a tab which can be used to visually manage the ETS tables on a given Erlang VM. If you have Erlang built with `wx` support, just call `observer:start()` and select the table viewer tab. On older Erlang releases, `observer` did not exist and you may have to use the now deprecated `tv` application instead (`tv:start()`).

DETS

DETS is a disk-based version of ETS, with a few key differences.

There are no longer `ordered_set` tables, there is a disk-size limit of 2GB for DETS files, and operations such as `prev/1` and `next/1` are not nearly as safe or fast.

Starting and stopping tables has changed a bit. A new database table is created by calling `dets:open_file/2`, and is closed by doing `dets:close/1`. The table can later be re-opened by calling `dets:open_file/1`.

Otherwise, the API is nearly the same, and it is thus possible to have a very simple way to handle writing and looking for data inside of files.

Don't Drink Too Much Kool-Aid:

DETS risks being slow as it is a disk-only database. It is possible you might feel like coupling ETS and DETS tables into a somewhat efficient database that stores both in RAM and on disk.

If you feel like doing so, it might be a good idea to look into *Mnesia* as a database, which does exactly the same thing, while adding support for sharding, transactions, and distribution.

A Little Less Conversation, A Little More Action Please



Following this rather long section title (and long previous sections), we'll turn to the practical problem that brought us here in the first place: updating regis so that it uses ETS and gets rid of a few potential bottlenecks.

Before we get started, we have to think of how we're going to handle operations, and what is safe and unsafe. Things that should be safe are those that modify nothing and are limited to one query (not 3-4 over time). They can be done by anyone at any time. Everything else that has to do with writing to a table, updating records, deleting them, or reading in a way that requires consistency over many requests are to be considered unsafe.

Because ETS has no transactions whatsoever, all unsafe operations should be performed by the process that owns the table. The safe ones should be allowed to be public, done outside of the owner process. We'll keep this in mind as we update regis.

The first step will be to make a copy of regis-1.0.0 as regis-1.1.0. I'm bumping the second number and not the third one here because our changes shouldn't break the existing interface, are technically not bugfixes, and so we're only going to consider it to be a feature upgrade.

In that new directory, we'll need to operate only on `regis_server.erl` at first: we'll keep the interface intact so all the rest, in terms of structure, should not need to change too much:

```
%%% The core of the app: the server in charge of tracking processes.  
-module(regis_server).  
-behaviour(gen_server).  
-include_lib("stdlib/include/ms_transform.hrl").  
  
-export([start_link/0, stop/0, register/2, unregister/1, whereis/1,
```

```

    get_names/0]).

-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
        code_change/3, terminate/2]).


%%%%%%%
%%% INTERFACE %%%
%%%%%%%

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

stop() ->
    gen_server:call(?MODULE, stop).

%% Give a name to a process
register(Name, Pid) when is_pid(Pid) ->
    gen_server:call(?MODULE, {register, Name, Pid}).

%% Remove the name from a process
 unregister(Name) ->
    gen_server:call(?MODULE, {unregister, Name}).

%% Find the pid associated with a process
whereis(Name) -> ok.

%% Find all the names currently registered.
get_names() -> ok.

```

For the public interface, only `whereis/1` and `get_names/0` will change and be rewritten. That's because, as mentioned earlier, those are single-read safe operations. The rest will require to be serialized in the process owning the table. That's it for the API so far. Let's head for the inside of the module.

We're going to use an ETS table to store stuff, so it makes sense to put that table into the `init` function. Moreover, because our `whereis/1` and `get_names/0` functions will give public access to the table (for speed reasons), naming the table will be necessary for it to be accessible to the outside world. By naming the table, much like what happens when we name processes, we can hardcode the name in the functions, compared to needing to pass an id around.

```

%%%%%%%
%%% GEN_SERVER CALLBACKS %%%
%%%%%%

init([]) ->
    ?MODULE = ets:new(?MODULE, [set, named_table, protected]),
    {ok, ?MODULE}.

```

The next function will be `handle_call/3`, handling the message `{register, Name, Pid}` as defined in `register/2`.

```

handle_call({register, Name, Pid}, _From, Tid) ->
    %% Neither the name or the pid can already be in the table

```

```

%% so we match for both of them in a table-long scan using this.
MatchSpec = ets:fun2ms(fun({N,P,_Ref}) when N==Name; P==Pid -> {N,P} end),
case ets:select(Tid, MatchSpec) of
    [] -> % free to insert
        Ref = erlang:monitor(process, Pid),
        ets:insert(Tid, {Name, Pid, Ref}),
        {reply, ok, Tid};
    [{Name,_}|_] -> % maybe more than one result, but name matches
        {reply, {error, name_taken}, Tid};
    [{_,Pid}|_] -> % maybe more than one result, but Pid matches
        {reply, {error, already_named}, Tid}
end;

```

This is by far the most complex function in the module. There are three basic rules to respect:

1. A process cannot be registered twice
2. A name cannot be taken twice
3. A process can be registered if it doesn't break rules 1 and 2

This is what the code above does. The match specification derived from `fun({N,P,_Ref}) when N==Name; P==Pid -> {N,P} end` will look through the whole table for entries that match either the name or the pid that we're trying to register. If there's a match, we return both the name and pids that were found. This may be weird, but it makes sense to want both when we look at the patterns for the case ... of after that.

The first pattern means nothing was found, and so insertions are good. We monitor the process we have registered (to unregister it in case of failure) and then add the entry to the table. In case the name we are trying to register was already in the table, the pattern `[{Name,_}|_]` will take care of it. If it was the Pid that matched, then the pattern `[{_,Pid}|_]` will take care of it. That's why both values are returned: it makes it simpler to match on the whole tuple later on, not caring which of them matched in the match spec. Why is the pattern of the form `[Tuple|_]` rather than just `[Tuple]`? The explanation is simple enough. If we're traversing the table looking for either Pids or names that are similar, it is possible the list return will be `[{NameYouWant, SomePid},{SomeName,PidYouWant}]`. If that happens, then a pattern match of the form `[Tuple]` will crash the process in charge of the table and ruin your day.

Oh yeah, don't forget to add the `-include_lib("stdlib/include/ms_transform.hrl")`. in the module, otherwise, `fun2ms` will die with a weird error message:

```

** {badarg,{ets,fun2ms,
    [function,called,with,real,'fun',should,be,transformed,with,
     parse_transform,'or',called,with,a,'fun',generated,in,the,
     shell]}}

```

That's what happens when you forget the include file. Consider yourself warned. Look before crossing the streets, don't cross the streams, and don't forget your include files.

The next bit to do is when we ask to manually unregister a process:

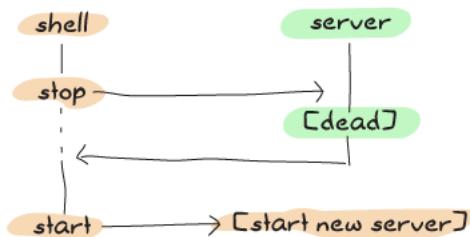
```
handle_call({unregister, Name}, _From, Tid) ->
    case ets:lookup(Tid, Name) of
        [{Name, _Pid, Ref}] ->
            erlang:demonitor(Ref, [flush]),
            ets:delete(Tid, Name),
            {reply, ok, Tid};
        [] ->
            {reply, ok, Tid}
    end;
```

If you looked at the old version of the code, this is still similar. The idea is simple: find the monitor reference (with a lookup on the name), cancel the monitor, then delete the entry and keep going. If the entry's not there, we pretend we deleted it anyway and everybody's going to be happy. Oh, how dishonest we are.

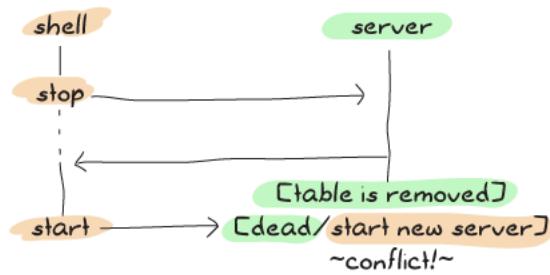
Next bit is about stopping the server:

```
handle_call(stop, _From, Tid) ->
    %% For the sake of being synchronous and because emptying ETS
    %% tables might take a bit longer than dropping data structures
    %% held in memory, dropping the table here will be safer for
    %% tricky race conditions, especially in tests where we start/stop
    %% servers a lot. In regular code, this doesn't matter.
    ets:delete(Tid),
    {stop, normal, ok, Tid};
handle_call(_Event, _From, State) ->
    {noreply, State}.
```

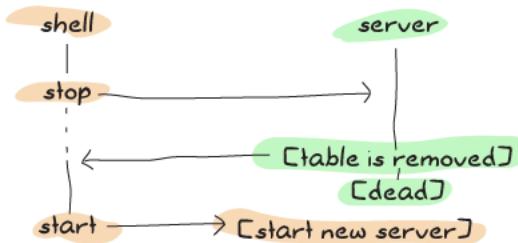
As the comments in the code say, we could have been fine just ignoring the table and letting it be garbage collected. However, because the test suite we have written for last chapter starts and stops the server all the time, delays can be a bit dangerous. See, this is what the timeline of the process looks like with the old one:



And here's what sometimes happens with the new one:



By using the scheme above, we're making it a lot more unlikely for errors to happen by doing more work in the synchronous part of the code:



If you don't plan on running the test suite very often, you can just ignore the whole thing. I've decided to show it to avoid nasty surprises, although in a non-test system, this kind of edge case should very rarely occur.

Here's the rest of the OTP callbacks:

```

handle_cast(_Event, State) ->
    {noreply, State}.

handle_info({'DOWN', Ref, process, _Pid, _Reason}, Tid) ->
    ets:match_delete(Tid, {'_', '_', Ref}),
    {noreply, Tid};

handle_info(_Event, State) ->
    {noreply, State}.

code_change(_OldVsn, State, _Extra) ->
    {ok, State}.

terminate(_Reason, _State) ->
    ok.

```

We don't care about any of them, except receiving a DOWN message, meaning one of the processes we were monitoring died. When that happens, we delete the entry based on the reference we have in the message, then move on.

You'll notice that `code_change/3` could actually work as a transition between the old `regis_server` and the new `regis_server`. Implementing this function is left as an exercise to the reader. !

always hate books that give exercises to the reader without solutions, so here's at least a little pointer so I'm not just being a jerk like all the other writers out there: you have to take either of the two gb_trees from the older version, and use gb_trees:map/2 or the gb_trees iterators to populate a new table before moving on. The downgrade function can be written by doing the opposite.

All that's left to do is fix the two public functions we have left unimplemented before. Of course, we could write a %% TODO comment, call it a day and go drink until we forget we're programmers, but that would be a tiny bit irresponsible. Let's fix stuff:

```
%% Find the pid associated with a process
whereis(Name) ->
    case ets:lookup(?MODULE, Name) of
        [{Name, Pid, _Ref}] -> Pid;
        [] -> undefined
    end.
```

This one looks for a name, returns the *Pid* or undefined depending on whether the entry has been found or not. Note that we do use regis_server (?MODULE) as the table name there; that's why we made it protected and named in the first place. For the next one:

```
%% Find all the names currently registered.
get_names() ->
    MatchSpec = ets:fun2ms(fun({Name, _, _}) -> Name end),
    ets:select(?MODULE, MatchSpec).
```

We use fun2ms again to match on the *Name* and keep only that. Selecting from the table will return a list and do what we need.

That's it! You can run the test suite in test/ to make things go:

```
$ erl -make
...
Recompile: src/regis_server
$ erl -pa ebin
...
1> eunit:test(regis_server).
All 13 tests passed.
ok
```

Hell yes. I think we can consider ourselves pretty good at ETS'ing now.

You know what would be really nice to do next? Actually exploring the distributed aspects of Erlang. Maybe we can bend our minds in a few more twisted ways before being done with the Erlang beast. Let's see.

Distribunomicon

Alone in the Dark

Oh hi! Please have a seat. I was expecting you. When you first heard of Erlang, there were two or three attributes that likely attracted you. Erlang is a functional language, it has great semantics for concurrency, and it supports distribution. We've now seen the first two attributes, spent time exploring a dozen more you possibly didn't expect, and we're now at the last big thing, distribution.

We've waited quite a while before getting here because it's not exactly useful to get distributed if we can't make things work locally in the first place. We're finally up to the task and have come a long way to get where we are. Like almost every other feature of Erlang, the distributed layer of the language was first added in order to provide fault tolerance. Software running on a single machine is always at risk of having that single machine dying and taking your application offline. Software running on many machines allows easier hardware failure handling, if, and only if the application was built correctly. There is indeed no benefit regarding fault-tolerance if your application runs on many servers, but cannot deal with one of them being taken down.



See, distributed programming is like being left alone in the dark, with monsters everywhere. It's scary, you don't know what to do or what's coming at you. Bad news: distributed Erlang is still leaving you alone in the dark to fight the scary monsters. It won't do any of that kind of hard work for you. Good news: instead of being alone with nothing but pocket change and a poor sense of aim to kill the monsters, Erlang gives you a flashlight, a machete, and a pretty kick-ass mustache to feel more confident (this also applies to female readers).

That's not especially due to how Erlang is written, but more or less due to the nature of distributed software. Erlang will write the few basic building blocks of distribution: ways to have many nodes (virtual machines) communicating with each other, serializing and deserializing data in communications, extending the concepts of multiple processes to many nodes, ways to monitor network failures, and so on. It will, however, not provide solutions to software-specific problems such as "what happens when stuff crashes."

This is the standard 'tools, not solutions' approach seen before in OTP; you rarely get full-blown software and applications, but you get many components to build systems with. You'll have tools that tell you when parts of the system go up or down, tools to do a bunch of stuff over the network, but hardly any silver bullet that takes care of fixing things for you.

Let's see what kind of flexing we can do with these tools.

This is my Boomstick

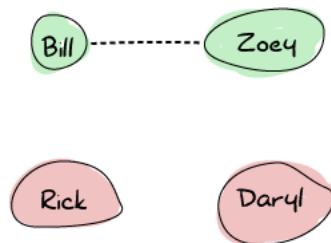
To tackle all these monsters in the dark, we've been granted a very useful thing: pretty complete network transparency.

An instance of an Erlang virtual machine that is up and running, ready to connect to other virtual machines is called a *node*. Whereas some languages or communities will consider a server to be a node, in Erlang, each VM is a node. You can thus have 50 nodes running on a single computer, or 50 nodes running on 50 computers. It doesn't really matter.

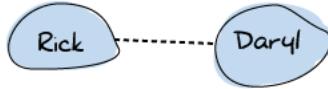
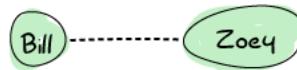
When you start a node, you give it a name and it will connect to an application called *EPMD* (Erlang Port Mapper Daemon), which will run on each of the computers which are part of your Erlang cluster. EPMD will act as a name server that lets nodes register themselves, contact other nodes, and warn you about name clashes if there are any.

From this point on, a node can decide to set up a connection to another one. When it does so, both nodes automatically start monitoring each other, and they can know if the connection is dropped, or if a node disappears. More importantly, when a new node joins another node which is already part of a group of nodes connected together, the new node gets connected to the entire group.

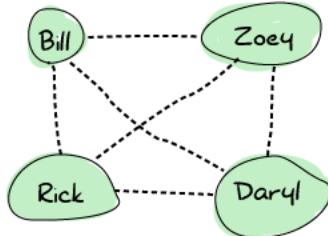
Let's take the idea of a bunch of survivors during a zombie outbreak to illustrate how Erlang nodes set up their connections. We've got Zoey, Bill, Rick, and Daryl. Zoey and Bill know each other and communicate on the same frequency on walkie-talkies. Rick and Daryl are each on their own:



Now let's say Rick and Daryl meet on their way to a survivor camp. They share their walkie-talkie frequency and can now stay up to date with each other before splitting ways again.



At some point, Rick meets Bill. Both are pretty happy about that, and so they decide to share frequencies. At this point, the connections spread and the final graph now looks like this:



That means that any survivor can contact any other directly. This is useful because in the event of the death of any survivor, nobody's left isolated. Erlang nodes are set up in this exact way: everyone connects to everyone.

Don't Drink Too Much Kool-Aid:

This way of doing things, while nice for some fault-tolerance reasons, has a pretty bad drawback in how much you can scale. It will be hard to have hundreds and hundreds of nodes part of your Erlang cluster simply because of how many connections and how much chatter is required. In fact, you will require one port per node you're connecting to.

If you were planning on using Erlang to do that kind of heavy setup, please read on in this chapter where we see why things are that way, and what might be done to go around the problem, if possible at all.

Once the nodes are connected together, they remain fully independent: they keep their own process registry, their own ETS tables (with their own names for tables), and the modules they load are independent from each other. A connected node that crashes won't bring down the nodes it's connected to.

Connected nodes can then start exchanging messages. The distribution model of Erlang was designed so that local processes can contact remote processes and send them regular messages. How is this possible if nothing is shared and all the process registries are unique? As we'll see later when we get into the specifics of distribution, there is a way to access registered processes on a particular node. From that point on, a first message can be sent.

Erlang messages are going to be serialized and unserialized automatically for you in a transparent manner. All data structures, including pids, will work the same remotely and locally. This means that we can send pids over the network, and then communicate with them, sending messages, etc. Even better than that, links and monitors can be set up across the network if you can access the pids!

So if Erlang's doing so much to make everything transparent, why am I saying it's only giving us a machete, a flashlight, and a mustache?

Fallacies of Distributed Computing

Much like a machete is meant to kill only a given type of monster, Erlang's tools are meant to handle only some kinds of distributed computing. To understand the tools Erlang gives us, it will be useful to first have an idea of what kind of landscape exists in the distributed world, and which assumptions Erlang makes in order to provide fault tolerance.

Some very smart guys took their time in the last few decades to categorize the kind of stuff that goes wrong with distributed computing. They came up with 8 major assumptions people make that ends up biting them in the ass later, some of which Erlang's designers made for various reasons.

The Network is Reliable

The first fallacy of distributed computing is assuming that the application can be distributed over the network. That's kind of weird to say, but there will be plenty of times where the network will go down for annoying reasons: power failures, broken hardware, someone tripping a cord, vortex to other dimensions engulfing mission-critical components, headcrabs infestation, copper theft, etc.

One of the biggest errors you can make, therefore, is to think you can reach remote nodes and talk to them. This is somewhat possible to handle by adding more hardware and gaining redundancy so that if some hardware fails, the application can still be reached somewhere else. The other thing to do is to be ready to suffer a loss of messages and requests, to be ready for things becoming unresponsive. This is especially true when you depend on some kind of third party service that's no longer there, while your own software stack keeps working well.

Erlang doesn't have any special measures to deal with this, as it's usually something where decisions made will be application-specific. After all, who else but you can know how important a specific component will be? Still, you're not totally alone as a distributed Erlang node will be able to detect other nodes getting disconnected (or becoming unresponsive). There are specific functions to monitor nodes, and links and monitors will also be triggered upon a disconnection.

Even with this, the best thing Erlang has for itself in this case is its asynchronous communication mode. By sending messages asynchronously and forcing developers to

send a reply back when things work well, Erlang pushes for all message passing activities to intuitively handle failure. If the process you're talking to is on a node that disappears due to some network failure, we handle it as naturally as any local crash. This is one of the many reasons why Erlang is said to scale well (scaling in performance, but also in design).

Don't Drink Too Much Kool-Aid:

Linking and monitoring across nodes can be dangerous. In the case of a network failure, all remote links and monitors are triggered at once. This might then generate thousands and thousands of signals and messages to various processes, which puts a heavy and unexpected load on the system.

Preparing for an unreliable network also means preparing for sudden failures and making sure your system doesn't get crippled by part of the system suddenly disappearing.

There is no Latency

One of the double-edged aspects of seemingly good distribution systems is that they often end up hiding the fact that the function calls you are making are remote. While you expect some function calls to be really fast, doing them over the network isn't the same at all. It's the difference between ordering a pizza from within the pizzeria and getting one delivered from another city to your house. While there will always be a basic wait time, in one case your pizza might be delivered cold because it just took too long.

Forgetting that network communications make things slower even for really small messages can be a costly error if you always expect really fast results. Erlang's model treats us well there. Because of the way we set up our local applications with isolated processes, asynchronous messages, timeouts and always thinking of the possibility for processes to fail, there is very little adaptation required to go distributed: the timeouts, links, monitors and asynchronous patterns remain the same and still are as reliable. We always expected that kind of problem from the beginning and so Erlang implicitly doesn't assume there is no latency.

You, however, might make that assumption in your design and expect replies faster than realistically possible. Just keep an eye open.

Bandwidth is Infinite

Although network transfers are getting faster and faster all the time, and that generally speaking, each byte transferred over the network is cheaper as time goes, it is risky to assume that sending copious amounts of data is simple and easy.

Generally speaking, because of how we build applications locally, we won't have too many problems with that in Erlang. Remember, one good trick is to send messages about what is happening rather than moving new state around ('Player X found item Y' rather than sending Player X's entire inventory over and over again).

If, for some reason, you need to be sending large messages, be extremely careful. The way Erlang distribution and communication works over many nodes is especially sensitive to large messages. If two nodes are connected together, all their communications will tend to happen over a single TCP connection. Because we generally want to maintain message ordering between two processes (even across the network), messages will be sent sequentially over the connection. That means that if you have one very large message, you might be blocking the channel for all the other messages.

Worse than that, Erlang knows whether nodes are alive or not by sending a thing called *heartbeats*. Heartbeats are small messages sent at a regular interval between two nodes basically saying "I'm still alive, keep on keepin' on!". They're like our Zombie survivors routinely pinging each other with messages; "Bill, are you there?" And if Bill never replies, then you might assume he's dead (our out of batteries) and he won't get your future communications. Anyway, heartbeats are sent over the same channel as regular messages.

The problem is that a large message can thus hold heartbeats back. Too many large messages keeping heartbeats at bay for too long and either of the nodes will eventually assume the other is unresponsive and disconnect from each other. That's bad. In any case, the good Erlang design lesson to keep this from happening is to keep your messages small. Everything will be better that way.

The Network is Secure

When you get distributed, it's often very dangerous to believe that everything is safe, that you can trust messages you receive. It can be simple things like someone unexpected fabricating messages and sending them to you, someone intercepting packets and modifying them (or looking at sensitive data), or in the worst case, someone being able to take over your application or the system it runs on.

In the case of distributed Erlang, this is sadly an assumption that was made. Here is what Erlang's security model looks like:

* this space intentionally left blank *

Yep. This is because Erlang distribution was initially meant for fault tolerance and redundancy of components. In the old days of the language, back when it was used for telephone switches and other telecommunication applications, Erlang would often be deployed on hardware running in the weirdest places — very remote locations with weird conditions (engineers sometimes had to attach servers to the wall to avoid wet ground, or install custom heating systems in the woods in order for the hardware to run at optimal temperatures). In these cases, you had failover hardware part of the same physical location as the main one. This is often where distributed Erlang would run, and it explains why Erlang designers assumed a safe network to operate with.

Sadly, this means that modern Erlang applications can rarely be clustered across different data centers. In fact, it isn't recommended to do so. Most of the time, you will want your system to be based on many smaller, walled off clusters of Erlang nodes, usually located in single locations. Anything more complex will need to be left to the developers: either switching to SSL, implementing their own high level communication layer, tunneling over secure channels, or reimplementing the communication protocol between nodes. Pointers on how to do so exist in the ERTS user guide, in How to implement an alternative carrier for the Erlang distribution. More details on the distribution protocol is contained in Distribution Protocol. Even in these cases, you have to be pretty careful, because someone gaining access to one of the distributed nodes then has access to all of them, and can run any command they can.

Topology Doesn't Change

When first designing a distributed application made to run on many servers, it is possible that you will have a given number of servers in mind, and maybe a given list of host names. Maybe you will design things with specific IP addresses in mind. This can be a mistake. Hardware dies, operations people move servers around, new machines are added, some are removed. The topology of your network will constantly change. If your application works with any of these topological details hard-coded, then it won't easily handle these kinds of changes in the network.

In the case of Erlang, there is no explicit assumption made in that way. However, it is very easy to let it creep inside your application. Erlang nodes all have a name and a host name,

and they can constantly be changing. With Erlang processes, you not only have to think about how the process is named, but also about where it is now located in a cluster. If you hard code both the names and hosts, you might be in trouble at the next failure. Don't worry too much though, as we'll later see a few interesting libraries that let us forget about node names and topology in general, while still being able to locate specific processes.

There is Only One Administrator

This is something a distribution layer of a language or library can't prepare you for, no matter what. The idea of this fallacy is that you do not always have only one main operator for your software and its servers, although it might be designed as if there were only one. If you decide to run many nodes on a single computer, then you might never have to care about this fallacy. However, if you get to run stuff across different locations, or a third party depends on your code, then you have to take care.

Things to pay attention to include giving others tooling to diagnose problems on your system. Erlang is somewhat easy to debug when you can manipulate a VM manually – you can even reload code on the fly if you need to, after all. Someone who cannot access your terminal and sit in front of the node will need different facilities to operate though.

Another aspect of this fallacy is that things like restarting servers, moving instances between data centers, or upgrading parts of your software stack isn't necessarily something only one person or a single team controls. In very large software projects, it is in fact very likely that many teams, or even many different software companies, take charge of different parts of a greater system.

If you're writing protocols for your software stack, being able to handle many versions of that protocol might be necessary depending on how fast or slow your users and partners are to upgrade their code. The protocol might contain information about its versioning from the beginning, or be able to change halfway through a transaction, depending on your needs. I'm sure you can think of more examples of things that can go wrong.

Transport Cost is Zero

This is a two-sided assumption. The first one relates to the cost of transporting data in terms of time, and the second one is related to the cost of transporting data in terms of money.

The first case assumes that doing things like serializing data is nearly free, very fast, and doesn't play a big role. In reality, larger data structures take longer to be serialized than small ones, and then need to be unserialized on the other end of the wire. This will be true no matter what you carry across the network. Small messages will help reduce how noticeable the effect of this is.

The second aspect of assuming transport cost is zero has to do with how much it costs to carry data around. In modern server stacks, memory (both in RAM and on disk) is often cheap compared to the cost of bandwidth, something you have to pay for continuously,

unless you own the whole network where things run. Optimizing for fewer requests with smaller messages will be rewarding in this case.

For Erlang, due to its initial use cases, no special care has been taken to do things like compress messages going cross-node (although the functions for it already exist). Instead, the original designers chose to let people implement their own communication layer if they required it. The responsibility is thus on the programmer to make sure small messages are sent and other measures are taken to minimize the costs of transporting data.

The Network is Homogeneous

This last assumption is about thinking that all components of a networked application will speak the same language, or will use the same formats to operate together.

For our zombie survivors, this can be a question of not assuming that all survivors will always speak English (or good English) when they lay their plans, or that a word will hold different meanings to different people.



In terms of programming, this is usually about not relying on closed standards, but using open ones instead, or being ready to switch from one protocol to another one at any point in time. When it comes to Erlang, the distribution protocol is entirely public, but all Erlang nodes assume that people communicating with them speak the same language. Foreigners trying to integrate themselves to an Erlang cluster either have to learn to speak Erlang's protocol, or Erlang apps need some kind of translation layer for XML, JSON, or whatever.

If it quacks like a duck and walks like a duck, then it must be a duck. That's why we have things like C-nodes. C-nodes (or nodes in other languages than C) are built on the idea that any language and application can implement Erlang's protocol and then pretend it is an Erlang node in a cluster.

Another solution for data exchange is to use something called BERT or BERT-RPC. This is an exchange format like XML or JSON, but specified as something similar to the Erlang External Term Format.

In short, you always have to be careful for the following points:

- You shouldn't assume the network is reliable. Erlang doesn't have any special measure for that except detecting that something went wrong for you (although that's not too bad as a feature)
- The network might be slow, from time to time. Erlang gives asynchronous mechanisms and knows about it, but you have to be careful so your application doesn't go against this and ruin it.
- Bandwidth isn't infinite. Small, descriptive messages help respect this.
- The network isn't secure, and Erlang doesn't have anything to offer by default for this.
- The topology of the network can change. No explicit assumption is made by Erlang, but you might make some about where things are and how they're named.
- You (or your organization) only rarely fully control the structure of things. Parts of your system may be outdated, use different versions, be restarted or down when you don't expect it.
- Transporting data has costs. Again, small, short messages help.
- The network is heterogeneous. Not everything is the same, and data exchange should rely on well-documented formats.

Note: The fallacies of distributed computing were introduced in Fallacies of Distributed Computing Explained by Arnon Rotem-Gal-Oz

Dead or Dead Alive

Understanding the fallacies of distributed computing should have partially explained why we're fighting monsters in the dark, but with better tools. There are still a lot of issues and things left for us to do. Many of them are design decisions to be careful about (small messages, reducing communication, etc.) regarding the fallacies above. The most problematic issue has to do with nodes dying or the network being unreliable. This one is especially nasty because there is no good way to know whether something is dead or alive (without being able to contact it).

Let's get back to Bill, Zoey, Rick and Daryl, our 4 Zombie apocalypse survivors. They all met at a safe house, spent a few days resting in there, eating whatever canned food they could find. After a while, they had to move out and split across town to find more resources. They've set a rendez-vous point in a small camp on the limits of the small town they're in.

During the expedition they keep contact by talking with the walkie-talkies. They announce what they found, clear paths, maybe they find new survivors.

Now suppose that at some point between the safe house and the rendez-vous point, Rick tries to contact his comrades. He manages to call Bill and Zoey, talk to them, but Daryl isn't reachable. Bill and Zoey can't contact him either. The problem is that there is absolutely no way to know if Daryl has been devoured by zombies, if his battery is dead, if he's asleep or if he's just underground.

The group has to decide whether to keep waiting for him, keep calling for a while, or assume he's dead and move forward.

The same dilemma exists with nodes in a distributed system. When a node becomes unresponsive, is it gone because of a hardware failure? Did the application crash? Is there congestion on the network? Is the network down? In some cases, the application is not running anymore and you can simply ignore that node and continue what you're doing. In other cases, the application is still running on the isolated node; from that node's perspective, everything else is dead.

Erlang made the default decision of considering unreachable nodes as dead nodes, and reachable nodes as alive. This is a pessimistic approach that makes sense if you want to very quickly react to catastrophic failures; it assumes that the network is generally less likely to fail than the hardware or the software in the system, which makes sense considering how Erlang was used originally. An optimistic approach (which assumes nodes are still alive) could delay crash-related measures because it assumes that the network is more likely to fail than hardware or the software, and thus have the cluster wait longer for the reintegration of disconnected nodes.

This raises a question. In a pessimistic system, what happens when the node we thought dead suddenly comes back again and it turns out it never died? We're caught by surprise by a living dead node, which had a life of its own, isolated from the cluster in every way: data, connections, etc. There are some very annoying things that can happen.

Let's imagine for a moment that you have a system with 2 nodes in 2 different data centers. In that system, users have money in their account, with the full amount held on each node. Each transaction then synchronizes the data to all the other nodes. When all the nodes are fine, a user can keep spending money until his account is empty and then nothing can be sold anymore.

The software is chugging along fine, but at some point, one of the nodes gets disconnected from the other. There is no way to know if the other side is alive or dead. For all we care, both nodes could still be receiving requests from the public, but without being able to communicate with each other.

There are two general strategies that can be taken: stop all transactions, or don't. The risk of picking the first one is that your product becomes unavailable and you're losing money. The risk of the second one is that a user with \$1000 in his account now has two servers that can accept \$1000 of transactions, for a total of \$2000! Whatever we do, we risk losing money if we don't do things right.

Isn't there a way by which we could avoid the issue entirely by keeping the application available during netsplits, without having to lose data in between servers?



My Other Cap is a Theorem

A quick answer to the previous question is *no*. There is sadly no way to keep an application alive and correct at the same time during a netsplit.

This idea is known as the *CAP Theorem* (You might be interested in *You Can't Sacrifice Partition Tolerance* too). The CAP Theorem first states that there are three core attributes to all distributed systems that exist: Consistency, Availability, and Partition Tolerance.

Consistency

In the previous example, consistency would be having the ability to have the system, whether there are 2 or 1000 nodes that can answer queries, to see exactly the same amount of money in the account at a given time. This is something usually done by adding transactions (where all nodes must agree to making a change to a database as a single unit before doing so) or some other equivalent mechanism.

By definition, the idea of consistency is that all operations look as if they were completed as a single indivisible block even across many nodes. This is not in terms of time, but in terms of not having two different operations on the same piece of data modifying them in ways that gives multiple different values reported by system during these operations. It should be possible to modify a piece of data and not have to worry about other actors ruining your day by fiddling with it at the same time you do.

Availability

The idea behind availability is that if you ask the system for some piece of data, you're able to get a response back. If you don't get an answer back, the system isn't available to you. Note that a response that says "sorry I can't figure out results because I'm dead" isn't really a response, but only a sad excuse for it. There is no more useful information in this response than in no response at all (although academics are somewhat divided on the issue).

Note: an important consideration in the CAP theorem is that availability is only a concern to nodes that are *not dead*. A dead node cannot send responses because it can't receive queries in the first place. This isn't the same as a node that can't send a reply because a

thing it depends on is no longer there! If the node can't take requests, change data or return erroneous results, it isn't technically a threat to the balance of the system in terms of correctness. The rest of the cluster just has to handle more load until it comes back up and can be synchronized.

Partition Tolerance

This is the tricky part of the CAP theorem. Partition tolerance usually means that the system can keep on working (and contain useful information) even when parts of it can no longer communicate together. The whole point of partition tolerance is that the system can work with messages possibly being lost between components. The definition is a bit abstract and open-ended, and we'll see why.

The CAP Theorem basically specifies that in any distributed system, you can only have two of CAP: either CA, CP, or AP. There is no possible way to have all of them. This is both bad and good news. The bad news is that it's impossible to have everything always going well even with a failing network. The good news is that this is a theorem. If a customer asks you to provide all three of them, you will have the advantage of being able to tell them it is literally impossible to do, and won't have to lose too much time outside of explaining to them what the hell the CAP theorem is.

Of the three possibilities, one that we can usually dismiss is the idea of CA (Consistency + Availability). The reason for this is that the only time you would really want this is if you dare to say the network will never fail, or that if it does, it does as an atomic unit (if one thing fails, everything does at once).

Until someone invents a network and hardware that never fails, or has some way to make all parts of a system fail at once if one of them does, failure is going to be an option. Only two combinations of the CAP theorem remain: AP or CP. A system torn apart by a netsplit can either remain available or consistent, but not both.

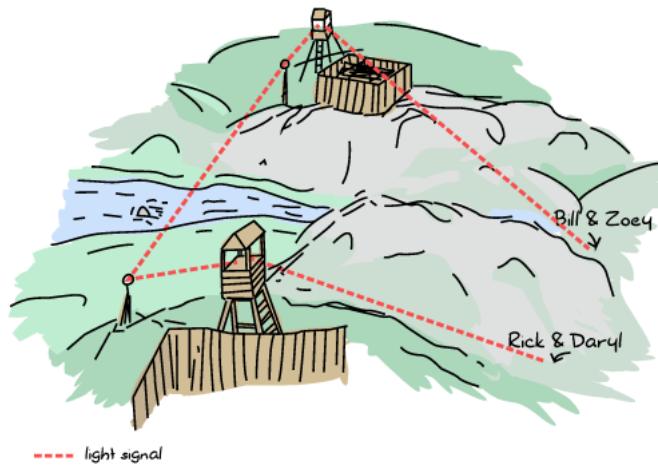
Note: some systems will choose to have neither 'A' or 'C'. In some cases of high performance, criteria such as throughput (how many queries you can answer at all) or latency (how fast can you answer queries) will bend things in a way such that the CAP theorem isn't about 2 attributes (CA, CP, or AP), but also about 2 and fewer attributes.

For our group of survivors, time passed and they fended off groups of undead for a good while. Bullets pierced brains, baseball bats shattered skulls and people bit were left behind. Bill, Zoey, Rick and Daryl's batteries eventually ran out and they were unable to communicate. As luck would have it, they all found two survivor colonies populated with computer scientists and engineers enamored with zombie survival. The colony survivors were used to the concepts of distributed programming and were used to communicating with light signals and mirrors with home-made protocols.

Bill and Zoey found the 'Chainsaw' colony while Rick and Daryl found the 'Crossbow' camp. Given that our survivors were the newest arrivals in their respective colonies, they were often delegated to go out in the wild, hunt for food and kill Zombies coming too close to the perimeters while the rest of people debated the merits of vim vs. emacs, the only war that couldn't die after a complete Zombie apocalypse.

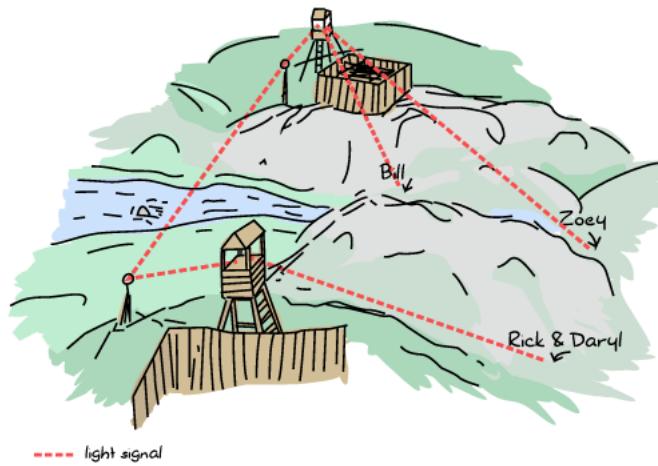
On their hundredth day there, our four survivors were sent to meet halfway across the camps to trade goods for each colony.

Before leaving, a rendez-vous point was decided by the chainsaw and crossbow colonies. If at any point in time the destination or meeting time were to change, Rick and Daryl could message the Crossbow colony or Zoey and Bill could message the Chainsaw colony. Then each colony would communicate the information to the other colony, which would forward the changes to the other survivors:



So knowing this, all four survivors left early on a Sunday morning for a long trip on foot, due to meet on Friday morning before dawn. Everything went fine (except the occasional skirmishes with dead people who had been alive for quite a while now).

Unfortunately, on Wednesday, heavy rain and increased zombie activity had Bill and Zoey separated, lost and delayed. The new situation looked a bit like this:



To make matters worse, after the rain, the usually clear sky between the two colonies got foggy and it became impossible for the Chainsaw computer scientists to communicate with the Crossbow people.

Bill and Zoey communicated their problems to their colony and asked to set new meeting times. This would have been alright without the fog, but now we've got the equivalent of a netsplit.

If both camps work under the Consistency + Partition Tolerance approach, they will just keep Zoey and Bill from setting a new meeting time. See, the CP approach is usually all about stopping modifications to the data so it remains consistent, and all survivors can still ask their respective camps for the date from time to time. They will just be denied the right to change it. Doing this will ensure that there is no way for some survivors to mess up the planned meeting time – any other survivor cut off from any contact could still meet there in time no matter what, independently.

If both camps instead picked Availability + Partition Tolerance, then survivors could have been allowed to change meeting dates. Each of the sides of the partitions would have their own version of the meeting data. So if Bill called for a new meeting for Friday night, the general state becomes:

Chainsaw: Friday night

Crossbow: Friday before dawn

As long as the split lasts, Bill and Zoey will get their information from Chainsaw only, and Rick and Daryl from Crossbow only. This lets part of the survivors reorganize themselves if needed.

The interesting problem here is how to handle the different versions of the data when the split is resolved (and the fog goes away). The CP approach to this is pretty straightforward: the data didn't change, there is nothing to do. The AP approach has more flexibility and problems to solve. Usually, different strategies are employed:

- *Last Write Wins* is a conflict resolution method where whatever the last update was is the one to be kept. This one can be tricky because in distributed settings, timestamps can be off or things can happen at exactly the same time.
- A winner can be picked randomly.
- More sophisticated methods to help reduce conflicts include time-based methods such as last write wins, but with relative clocks. Relative clocks do not work with absolute time values, but with incrementing values every time someone modifies a file. If you want to know more about this, read up on Lamport clocks or vector clocks.
- The onus of picking what to do with the conflict can be pushed back to the application (or in our case, to the survivors). The receiving end will just have to choose which of the conflicting entries is the right one. This is a bit what happens when you have merge conflicts with source control with SVN, Mercurial, Git, etc.

Which one's better? The way I've described things kind of led us to believe that we have the choice to be either fully AP or fully CP, like an on/off switch. In the real world, we can have various things like quorum systems where we turn this 'yes/no' question into a dial we can turn to choose how much consistency we want.

A quorum system works by a few very simple rules. You have N nodes in the system and require M of them to agree to modify the data to make it possible. A system with a relatively low consistency requirement could ask for only 15% of the nodes to be available to make a change. This means that in cases of splits, even small fragments of the network keep being able to modify the data. A higher consistency rating, set to maybe 75% of the nodes would mean that a larger part of the system needs to be present in order to make changes. In this situation, if a few of the nodes are isolated, they won't have the right to change the data. However, the major part of the system that's still interconnected can work fine.

By changing making the M value of required nodes up to N (the total number of nodes), you can have a fully consistent system. By giving M the value 1, you have a fully AP system, with no consistency guarantees.

Moreover, you could play with these values on a per-query basis: queries having to do with things of little importance (someone just logged on!) can have lower consistency requirements, while things having to do with inventory and money could require more consistency. Mix this in with different conflict resolution methods for each case and you can get surprisingly flexible systems.

Combined with all the different conflict resolution solutions available, a lot of options become available to distributed systems, but their implementation remains very complex. We won't use them in detail, but I think it's important to know what's available out there just to be aware of the different options available.

For now, we can stick to the basics of distributed computing with Erlang.

Setting up an Erlang Cluster

Except for the whole part about handling the fallacies of distributed computing, the hardest part about distributed Erlang is managing to set things up right in the first place. Connecting nodes together across different hosts is a special kind of pain. To avoid this, we'll usually try things out using many nodes on a single computer, which tends to make things easier.

As mentioned earlier, Erlang gives names to each of the nodes to be able to locate and contact them. The names are of the form `Name@Host`, where the host is based on available DNS entries, either over the network or in your computer's host files (`/etc/hosts` on OSX, Linux and other Unix-likes, `C:\Windows\system32\drivers\etc\hosts` for most Windows installs). All names need to be unique to avoid conflicts — if you try to start a node with the same name as another one on the same exact hostname, you'll get a pretty terrible crash message.

Before starting these shells to provoke a crash, we have to know a bit about the names. There are two types of names: short names and long names. Long names are based on fully qualified domain names (`aaa.bbb.ccc`), and many DNS resolvers consider a domain name to be fully qualified if they have a period (.) inside of it. Short names will be based on host names without a period, and are resolved going through your host file or through any possible DNS entry. Because of this, it is generally easier to set up a bunch of Erlang nodes together on a single computer using short names than long names. One last thing: because names need to be unique, nodes with short names cannot communicate with nodes that have long names, and the opposite is also true.

To pick between long and short names, you can start the Erlang VM with two different options: `erl -sname short_name@domain` or `erl -name long_name@some.domain`. Note that you can also start nodes with only the names: `erl -sname short_name` or `erl -name long_name`. Erlang will automatically attribute a host name based on your operating system's configuration. Lastly, you also have the option of starting a node with a name such as `erl -name name@127.0.0.1` to give a direct IP.

Note: Windows users should still use `werl` instead of `erl`. However, in order to start distributed nodes and giving them a name, the node should be started from the command line instead of clicking some shortcut or executable.

Let's start two nodes:

```
erl -sname ketchup
...
(ketchup@ferdmbp)1>
erl -sname fries
...
(fries@ferdmbp)1>
```

To connect `fries` with `ketchup` (and make a delicious cluster) go to the first shell and enter the following function:

```
(ketchup@ferdmbp)1> net_kernel:connect_node(fries@ferdmbp).
true
```

The `net_kernel:connect_node(NodeName)` function sets up a connection with another Erlang node (some tutorials use `net_adm:ping(Node)`, but I think `net_kernel:connect_node/1` sounds more serious and lends me credence!) If you see `true` as the result from the function call, congratulations, you're in distributed Erlang mode now. If you see `false`, then you're in for a world of hurt trying to get your network to play nice. For a very quick fix, edit your host files to accept whatever host you want. Try again and see if it works.

You can see your own node name by calling the BIF `node()` and see who you're connecting to by calling the BIF `nodes()`:

```
(ketchup@ferdmbp)2> node().
ketchup@ferdmbp
(ketchup@ferdmbp)3> nodes().
[fries@ferdmbp]
```

To get the nodes communicating together, we'll try with a very simple trick. Register each shell's process as shell locally:

```
(ketchup@ferdmbp)4> register(shell, self()).
true
(fries@ferdmbp)1> register(shell, self()).
true
```

Then, you'll be able to call the process by name. The way to do it is to send a message to `{Name, Node}`. Let's try this on both shells:

```
(ketchup@ferdmbp)5> {shell, fries@ferdmbp} ! {hello, from, self()}.
{hello,from,<0.52.0>}
(fries@ferdmbp)2> receive {hello, from, OtherShell} -> OtherShell ! <<"hey there!">> end.
<<"hey there!">>
```

So the message is apparently received, and we send something to the other shell, which receives it:

```
(ketchup@ferdmbp)6> flush().
Shell got <<"hey there!">>
ok
```

As you can see, we transparently send tuples, atoms, pids, and binaries without a problem. Any other Erlang data structure is fine too. And that's it. You know how to work with distributed Erlang! There is yet another BIF that might be useful: `erlang:monitor_node(NodeName, Bool)`. This function will let the process that calls it with `true` as a value for `Bool` receive a message of the format `{nodedown, NodeName}` if the node dies.

Unless you're writing a special library that relies on checking the life of other nodes, you will rarely need to use erlang:monitor_node/2. The reason for this is that functions like link/1 and monitor/2 still work across nodes.

If you set up the following from the fries node:

```
(fries@ferdmbp)3> process_flag(trap_exit, true).
false
(fries@ferdmbp)4> link(OtherShell).
true
(fries@ferdmbp)5> erlang:monitor(process, OtherShell).
#Ref<0.0.0.132>
```

And then kill the ketchup node, the fries' shell process should receive an 'EXIT' and monitor message:

```
(fries@ferdmbp)6> flush().
Shell got {'DOWN',#Ref<0.0.0.132>,process,<6349.52.0>,noconnection}
Shell got {'EXIT',<6349.52.0>,noconnection}
ok
```

And that's the kind of stuff you'll see. But hey, wait a minute there. Why the hell does the pid look like that? Am I seeing things right?

```
(fries@ferdmbp)7> OtherShell.
<6349.52.0>
```

What? Shouldn't this be <0.52.0>? Nope. See, that way of displaying a pid is just some kind of visual representation of what a process identifier is really like. The first number represents the node (where 0 means the process is coming from the current node), the second one is a counter, and the third one is a second counter for when you have so many processes created that the first counter is not enough. The true underlying representation of a pid is more like this:

```
(fries@ferdmbp)8> term_to_binary(OtherShell).
<<131,103,100,0,15,107,101,116,99,104,117,112,64,102,101,
 114,100,109,98,112,0,0,0,52,0,0,0,0,3>>
```

The binary sequence <<107,101,116,99,104,117,112,64,102,101,114,100,109,98,112>> is in fact a latin-1 (or ASCII) representation of <<"ketchup@ferdmbp">>, the name of the node where the process is located. Then we have the two counters, <<0,0,52>> and <<0,0,0>>. The last value (3) is some token value to differentiate whether the pid comes from an old node, a dead one, etc. That's why pids can be used transparently anywhere.

Note: Instead of killing a node to disconnect it, you may also want to try the BIF erlang:disconnect_node(Node) to get rid of the node without shutting it down.

Note: if you're unsure which node a Pid is coming from, you don't need to convert it to a binary to read the node name. Just call node(Pid) and the node where it's running on will be

returned as a string.

Other interesting BIFs to use are `spawn/2`, `spawn/4`, `spawn_link/2` and `spawn_link/4`. They work exactly like the other `spawn` BIFs except that these let you spawn functions on remote nodes. Try this from the ketchup node:

```
(ketchup@ferdmbp)6> spawn(fries@ferdmbp, fun() -> io:format("I'm on ~p~n", [node()])) end.  
I'm on fries@ferdmbp  
<6448.50.0>
```

This is essentially a remote procedure call: we can choose to run arbitrary code on other nodes, without giving ourselves more trouble than that! Interestingly, the function is running on the other node, but we receive the output locally. That's right, even output can be transparently redirected across the network. The reason for this is based on the idea of group leaders. Group leaders are inherited the same way whether they're local or not.

Those are all the tools you need in Erlang to be able to write distributed code. You have just received your machete, flashlight and mustache. You're at a level that would take a very long while to achieve with other languages without such a distribution layer. Now is the time to kill monsters. Or maybe first, we have to learn about the cookie monster.

Cookies



If you recall the beginning of the chapter, I had mentioned the idea that all Erlang nodes are set up as meshes. If someone connects to a node, it gets connected to all the other nodes. There are times where what you want to do is run different Erlang node clusters on the same piece of hardware. In these cases, you do not want to be accidentally connecting two Erlang node clusters together.

Because of this, the designers of Erlang added a little token value called a *cookie*. While documents like the official Erlang documentation put cookies under the topic of security, they're really not security at all. If it is, it has to be seen as a joke, because there's no way anybody serious considers the cookie a safe thing. Why? Simply because the cookie is a little unique value that must be shared between nodes to allow them to connect together. They're closer to the idea of user names than passwords and I'm pretty sure nobody would consider having a username (and nothing else) as a security feature. Cookies make way

more sense as a mechanism used to divide clusters of nodes than as an authentication mechanism.

To give a cookie to a node, just start it by adding a `-setcookie Cookie` argument to the command line. Let's try again with two new nodes:

```
$ erl -sname salad -setcookie 'myvoiceismypassword'  
...  
(salad@ferdmbp)1>  
  
$ erl -sname mustard -setcookie 'opensesame'  
...  
(mustard@ferdmbp)1>
```

Now both nodes have different cookies and they shouldn't be able to communicate together:

```
(salad@ferdmbp)1> net_kernel:connect_node(mustard@ferdmbp).  
false
```

This one has been denied. Not many explanations. However, if we look at the mustard node:

```
=ERROR REPORT==== 10-Dec-2011::13:39:27 ====  
** Connection attempt from disallowed node salad@ferdmbp **
```

Good. Now what if we did really want salad and mustard to be together? There's a BIF called `erlang:set_cookie/2` to do what we need. If you call `erlang:set_cookie(OtherNode, Cookie)`, you will use that cookie only when connecting to that other node. If you instead use `erlang:set_cookie(node(), Cookie)`, you'll be changing the node's current cookie for all future connections. To see the changes, use `erlang:get_cookie()`:

```
(salad@ferdmbp)2> erlang:get_cookie().  
myvoiceismypassword  
(salad@ferdmbp)3> erlang:set_cookie(mustard@ferdmbp, opensesame).  
true  
(salad@ferdmbp)4> erlang:get_cookie().  
myvoiceismypassword  
(salad@ferdmbp)5> net_kernel:connect_node(mustard@ferdmbp).  
true  
(salad@ferdmbp)6> erlang:set_cookie(node(), now_it_changes).  
true  
(salad@ferdmbp)7> erlang:get_cookie().  
now_it_changes
```

Fantastic. There is one last cookie mechanism to see. If you tried the earlier examples of this chapter, go look into your home directory. There should be a file named `.erlang.cookie` in there. If you read it, you'll have a random string that looks a bit like `PMIYERCHJZNZGSRJPVRK`. Whenever you start a distributed node without a specific command to give it a cookie, Erlang will create one and put it in that file. Then, every time you start a node again without specifying its cookie, the VM will look into your home directory and use whatever is in that file.

Remote Shells

One of the first things we've learned in Erlang was how to interrupt running code using ^G (CTRL + G). In there, we had seen a menu for distributed shells:

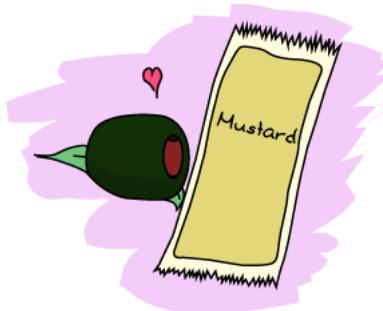
```
(salad@ferdmbp)1>
User switch command
--> h
c [nn]      - connect to job
i [nn]      - interrupt job
k [nn]      - kill job
j           - list all jobs
s [shell]    - start local shell
r [node [shell]] - start remote shell
q           - quit erlang
? | h       - this message
```

The r [node [shell]] option is the one we're looking for. We can start a job on the mustard node by doing as follows:

```
--> r mustard@ferdmbp
--> j
1 {shell,start,[init]}
2* {mustard@ferdmbp,shell,start,[]}
--> c
Eshell V5.8.4 (abort with ^G)
(mustard@ferdmbp)1> node().
mustard@ferdmbp
```

And there you have it. You can now use the remote shell the same way you would with a local one. There are a few differences with older versions of Erlang, where things like auto-completion no longer work. This way of doing things is still very useful whenever you need to change things on a node running with the -noshell option. If the -noshell node has a name, then you can connect to it to do admin-related things like reloading modules, debugging some code, and so on.

By using ^G again, you can go back to your original node. Be careful when you stop your session though. If you call q() or init:stop(), you'll be terminating the remote node!



Hidden Nodes

Erlang nodes can be connected by calling `net_kernel:connect_node/1`, but you have to be aware that pretty much any interaction between nodes will get them to set up a connection. Calling `spawn/2` or sending a message to a foreign Pid are going to automatically set up connections.

This might be rather annoying if you have a decent cluster and you want to connect to a single node to change a few things there. You wouldn't want your admin node to suddenly be integrated into the cluster, and having other nodes believing that they've got a new coworker to send tasks to. To do this, you could use the rarely-used `erlang:send(Dest, Message, [noconnect])` function, which sends a message without creating a connection, but this is rather error prone.

Instead, what you want to do is set up a node with the `-hidden` flag. Let's say you're still running the mustard and salad nodes. We'll start a third node, `olives` that will connect only to mustard (make sure the cookies are the same!):

```
$ erl -sname olives -hidden  
...  
(olives@ferdmbp)1> net_kernel:connect_node(mustard@ferdmbp).  
true  
(olives@ferdmbp)2> nodes().  
[]  
(olives@ferdmbp)3> nodes(hidden).  
[mustard@ferdmbp]
```

Ah ha! The node didn't connect to salad, and at first sight, it didn't connect with mustard either. However, calling `node(hidden)` shows that we do have a connection there! Let's see what the mustard node sees:

```
(mustard@ferdmbp)1> nodes().  
[salad@ferdmbp]  
(mustard@ferdmbp)2> nodes(hidden).  
[olives@ferdmbp]  
(mustard@ferdmbp)3> nodes(connected).  
[salad@ferdmbp,olives@ferdmbp]
```

Similar view, but now we add the `nodes(connected)` BIF that shows all connections, regardless of their type. The salad node will never see any connection to olives, unless especially told to connect there. One last interesting use of `nodes/1` is using `nodes(known)` which will show all nodes that the current node ever connected to.

With remote shells, cookies, and hidden nodes, managing distributed Erlang system becomes simpler.

The Walls are Made of Fire and the Goggles do Nothing

If you find yourself wanting to go through a firewall with distributed Erlang (and do not want to tunnel), you will likely want to open a few ports here and there for Erlang communication. If you want to do so, you will want to open up port 4369, the default port for EPMD. It's a good idea to use this one, because it's been officially registered for EPMD by Ericsson. This means that any standards-compliant operating-system you use will have that port free, ready for EPMD.

Then you will want to open a range of ports for connections between nodes. The problem is that Erlang just assigns random port numbers to inter-node connections. There are, however, two hidden application variables that let you specify a range within which ports can be assigned. The two values are `inet_dist_listen_min` and `inet_dist_listen_max` from the kernel application.

You could, as an example, start Erlang as `erl -name left_4_distribudead -kernel inet_dist_listen_min 9100 -kernel inet_dist_listen_max 9115` in order to set a range of 16 ports to be used for Erlang nodes. You could alternatively have a config file `ports.config` looking a bit like this:

```
[{kernel,[  
    {inet_dist_listen_min, 9100},  
    {inet_dist_listen_max, 9115}  
]}].
```

And then starting the Erlang node as `erl -name the_army_of_darknodes -config ports`. The variables will be set in the same way.

The Calls from Beyond

On top of all the BIFs and concepts we've seen, there are a few modules that can be used to help developers work with distribution. The first of these is `net_kernel`, which we used to connect nodes, and, as noted earlier, can be used to disconnect them.

It has some other fancy functionality, such as being able to transform a non-distributed node into a distributed one:

```
erl  
...  
1> net_kernel:start([romero, shortnames]).  
{ok,<0.43.0>}  
(romero@ferdmbp)2>
```

Where you can use either `shortnames` or `longnames` to define whether you want to have the equivalent of `-sname` or `-name`. Moreover, if you know a node is going to be sending large messages and thus might need a large heartbeat time between nodes, a third argument can be passed to the list. This gives `net_kernel:start([Name, Type, HeartbeatInMilliseconds])`. By default, the heartbeat delay is set to 15 seconds, or 15,000 milliseconds. After 4 failed heartbeats, a remote node is considered dead. The heartbeat delay multiplied by 4 is called the *tick time*.

Other functions of the module include `net_kernel:set_net_ticktime(S)` that lets you change the tick time of the node to avoid disconnections (where S is in seconds this time, and because it's a tick time, it must be 4 times the heartbeat delay!), and `net_kernel:stop()` to stop being distributed and go back to being a normal node:

```
(romero@ferdmbp)2> net_kernel:set_net_ticktime(5).
change_initiated
(romero@ferdmbp)3> net_kernel:stop().
ok
4>
```

The next useful module for distribution is `global`. The `global` module is a new alternative process registry. It automatically spreads its data to all connected nodes, replicates data there, handles node failures and supports different conflict resolution strategies when nodes get back online again.

You register a name by calling `global:register_name(Name, Pid)`, unregister with `global:unregister_name(Name)`. In case you want to do a name transfer without ever having it point to nothing, you can call `global:re_register_name(Name, Pid)`. You can find a process' id with `global:whereis_name(Name)`, and send a message to one by calling `global:send(Name, Message)`. There is everything you need. What's especially nice is that the names you use to register the processes can be *any* term at all.

A naming conflict will happen when two nodes get connected and both of them have two different processes sharing the same name. In these cases, `global` will kill one of them randomly by default. There are ways to override that behaviour. Whenever you register or re-register a name, pass a third argument to the function:

```
5> Resolve = fun(_Name,Pid1,Pid2) ->
5>   case process_info(Pid1,message_queue_len) > process_info(Pid2,message_queue_len) of
5>     true -> Pid1;
5>     false -> Pid2
5>   end
5> end.
#Fun<erl_eval.18.59269574>
6> global:register_name({zombie,12}, self(), Resolve).
yes
```

The `Resolve` function will pick the process with the most messages in its mailbox as the one to keep (it's the one the function returns the pid of). You could alternatively contact both processes and ask for who has the most subscribers, or only keep the first one to reply, etc. If the `Resolve` function crashes or returns something else than the pids, the process name is unregistered. For your convenience, the `global` module already defines three functions for you:

1. `fun global:random_exit_name/3` will kill a process randomly. This is the default option.

2. fun global:random_notify_name/3 will randomly pick one of the two processes as the one to survive, and it will send {global_name_conflict, Name} to the process that lost.
3. fun global:notify_all_name/3 it unregisters both pids, and sends the message {global_name_conflict, Name, OtherPid} to both processes and lets them resolve the issue themselves so they re-register again.



The global module has one downside in that it is often said to be rather slow to detect name conflicts and nodes going down. Otherwise it is a fine module, and it's even supported by behaviours. Just change all the gen_Something:start_link(...) calls that use local names ({local, Name}) to instead use {global, Name}, and then all calls and casts (and their equivalents) to use {global, Name} instead of just Name and things will be distributed.

The next module on the list is rpc, which stands for *Remote Procedure Call*. It contains functions that let you execute commands on remote nodes, and a few which facilitate parallel operations. To test these out, let's begin by starting two different nodes and connecting them together. I won't show the steps this time because I assume you now understand how this works. The two nodes are going to be cthulu and lovecraft.

The most basic rpc operation is rpc:call/4-5. It allows you to run a given operation on a remote node and get the results locally:

```
(cthulu@ferdmbp)1> rpc:call(lovecraft@ferdmbp, lists, sort, [[a,e,f,t,h,s,a]]).
[a,a,e,f,h,s,t]
(cthulu@ferdmbp)2> rpc:call(lovecraft@ferdmbp, timer, sleep, [10000], 500).
{badrpc,timeout}
```

As seen in this Call of the Cthulu node, the function with four arguments takes the form rpc:call(Node, Module, Function, Args). Adding a fifth argument gives a timeout. The rpc call will return whatever was returned by the function it ran, or {badrpc, Reason} in case of a failure.

If you've studied some distributed or parallel computing concepts before, you might have heard of promises. Promises are a bit like remote procedure calls, except that they are asynchronous. The rpc module lets us have this:

```
(cthulu@ferdmbp)3> Key = rpc:async_call(lovecraft@ferdmbp, erlang, node, []).
<0.45.0>
(cthulu@ferdmbp)4> rpc:yield(Key).
lovecraft@ferdmbp
```

By combining the result of the function `rpc:async_call/4` with the function `rpc:yield(Res)`, we can have asynchronous remote procedure calls and fetch the result later on. This is especially useful when you know the RPC you will make will take a while to return. Under these circumstances, you send it off, get busy doing other stuff in the mean time (other calls, fetching records from a database, drinking tea) and then wait on the results when there's absolutely nothing else left to do. Of course, you can do such calls on your own node if you need to:

```
(cthulu@ferdmbp)5> MaxTime = rpc:async_call(node(), timer, sleep, [30000]).  
<0.48.0>  
(cthulu@ferdmbp)6> lists:sort([a,c,b]).  
[a,b,c]  
(cthulu@ferdmbp)7> rpc:yield(MaxTime).  
... [long wait] ...  
ok
```

If by any chance you wanted to use the `yield/1` function with a timeout value, use `rpc:nb_yield(Key, Timeout)` instead. To poll for results, use `rpc:nb_yield(Key)` (which is equivalent to `rpc:nb_yield(Key,0)`):

```
(cthulu@ferdmbp)8> Key2 = rpc:async_call(node(), timer, sleep, [30000]).  
<0.52.0>  
(cthulu@ferdmbp)9> rpc:nb_yield(Key2).  
timeout  
(cthulu@ferdmbp)10> rpc:nb_yield(Key2).  
timeout  
(cthulu@ferdmbp)11> rpc:nb_yield(Key2).  
timeout  
(cthulu@ferdmbp)12> rpc:nb_yield(Key2, 1000).  
timeout  
(cthulu@ferdmbp)13> rpc:nb_yield(Key2, 100000).  
... [long wait] ...  
{value,ok}
```

If you don't care about the result, then you can use `rpc:cast(Node, Mod, Fun, Args)` to send a command to another node and forget about it.

The futures are yours, now! But wait, what if what we want is to call more than one node at a time? Let's add three nodes to our little cluster: `minion1`, `minion2` and `minion3`. Those are Cthulu's minions. When we want to ask them questions, we have to send 3 different calls, and when we want to give orders, we have to cast 3 times. That's pretty bad, and it doesn't scale with very large armies.

The trick is to use two RPC functions for calls and casts, respectively `rpc:multicall(Nodes, Mod, Fun, Args)` (with an optional `Timeout` argument) and `rpc:eval_everywhere(Nodes, Mod, Fun, Args)`:

```
(cthulu@ferdmbp)14> nodes().  
[lovecraft@ferdmbp, minion1@ferdmbp, minion2@ferdmbp, minion3@ferdmbp]
```

```
(cthulu@ferdmbp)15> rpc:multicall(nodes(), erlang, is_alive, []).  
{[true,true,true,true],[]}
```

This, right there, tells us that all four nodes are alive (and nobody was unavailable for an answer). The left side of the tuple is alive, the right side isn't. Yeah, `erlang:is_alive()` just returns whether the node it runs on is alive or not, which might look a bit weird. Yet again, remember that in a distributed setting, `alive` means 'can be reached', not 'is it running'. Then let's say Cthulu isn't really appreciative of its minions and decides to kill them, or rather, talk them into killing themselves. This is an order, and so it's cast. For this reason, we use `eval_everywhere/4` with a call to `init:stop()` on the minion nodes:

```
(cthulu@ferdmbp)16> rpc:eval_everywhere([minion1@ferdmbp, minion2@ferdmbp, minion3@ferdmbp], init, stop, [])  
abcast  
(cthulu@ferdmbp)17> rpc:multicall([lovecraft@ferdmbp, minion1@ferdmbp, minion2@ferdmbp, minion3@ferdmbp],  
{[true],[minion1@ferdmbp, minion2@ferdmbp, minion3@ferdmbp]})
```

When we ask again for who is alive, only one node remains, the Lovecraft node. The minions were obedient creatures. There are a few more interesting functions for RPC in there, but the core uses were covered here. If you want to know more, I suggest you comb through the documentation for the module.

Burying the Distribunomicon

Alright, that's it for most of the basics on distributed Erlang. There's a lot of things to think about, a lot of attributes to keep in mind. Whenever you have to develop a distributed application, ask yourself which of the distributed computing fallacies you could potentially run into (if any). If a customer asks you to build a system that handles netsplits while staying consistent *and* available, you know that you need to either calmly explain the CAP theorem or run away (possibly by jumping through a window, for a maximal effect).

Generally, applications where a thousand isolated nodes can do their job without communicating or depending on each other will provide the best scalability. The more inter-node dependencies created, the harder it becomes to scale, no matter what kind of distribution layer you have. This is just like zombies (no, really!). Zombies are terrifying because of how many of them there are, and how impossibly difficult to kill they can be as a group. Even though individual zombies can be very slow and far from menacing, a horde can do considerable damage, even if it loses many of its zombie members. Groups of human survivors can do great things by combining their intelligence and communicating together, but each loss they suffer is more taxing on the group and its ability to survive.

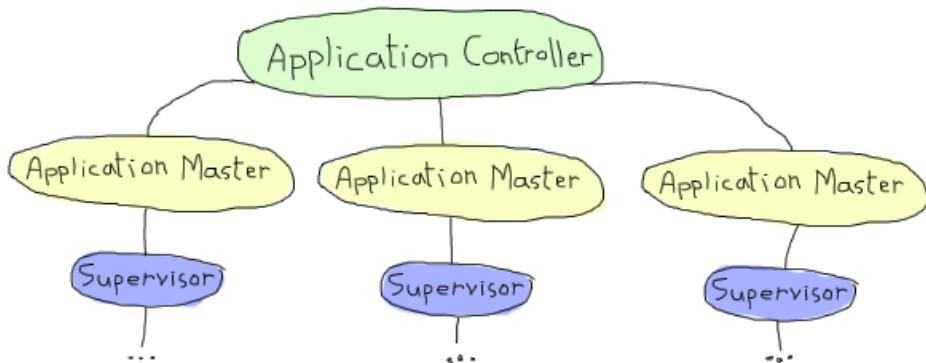
That being said, you've got the tools required to get going. The next chapter's going to introduce the concept of distributed OTP applications — something that provides a takeover and failover mechanism for hardware failures, but not general distribution; it's more like respawning your dead zombie than anything else.

Distributed OTP Applications

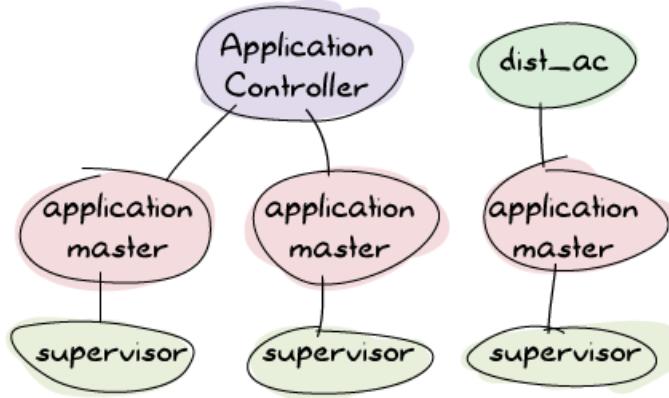
Although Erlang leaves us with a lot of work to do, it still provided a few solutions. One of these is the concept of *distributed OTP applications*. Distributed OTP applications, or just *distributed applications* when in the context of OTP, allow to define *takeover* and *failover* mechanisms. We'll see what that means, how that works, and write a little demo app to go with it.

Adding More to OTP

If you recall the chapter on OTP applications, we briefly saw the structure of an application as something using a central application controller, dispatching to application masters, each monitoring a top-level supervisor for an application:



In standard OTP applications, the application can be loaded, started, stopped or unloaded. In distributed applications, we change how things work; now the application controller shares its work with the *distributed application controller*, another process sitting next to it (usually called *dist_ac*):



Depending on the application file, the ownership of the application will change. A `dist_ac` will be started on all nodes, and all `dist_acs` will communicate together. What they talk about is not too relevant, except for one thing. As mentioned earlier, the four application statuses were being loaded, started, stopped, and unloaded; distributed applications split the idea of a started application into *started* and *running*.

The difference between both is that you could define an application to be global within a cluster. An application of this kind can only run on one node at a time, while regular OTP applications don't care about whatever's happening on other nodes.

As such a distributed application will be started on all nodes of a cluster, but only running on one.

What does this mean for the nodes where the application is started without being run? The only thing they do is wait for the node of the running application to die. This means that when the node that runs the app dies, another node starts running it instead. This can avoid interruption of services by moving around different subsystems.

Let's see what this means in more detail.

Taking and Failing Over

There are two important concepts handled by distributed applications. The first one is the idea of a *failover*. A failover is the idea described above of restarting an application somewhere else than where it stopped running.

This is a particularly valid strategy when you have redundant hardware. You run something on a 'main' computer or server, and if it fails, you move it to a backup one. In larger scale deployments, you might instead have 50 servers running your software (all at maybe 60–70% load) and expect the running ones to absorb the load of the failing ones. The concept of failing over is mostly important in the former case, and somewhat least interesting in the latter one.

The second important concept of distributed OTP applications is the *takeover*. Taking over is the act of a dead node coming back from the dead, being known to be more important than the backup nodes (maybe it has better hardware), and deciding to run the application again. This is usually done by gracefully terminating the backup application and starting the main one instead.

Note: In terms of distributed programming fallacies, distributed OTP applications assume that when there is a failure, it is likely due to a hardware failure, and not a netsplit. If you deem netsplits more likely than hardware failures, then you have to be aware of the possibility that the application is running both as a backup and main one, and that funny things could happen when the network issue is resolved. Maybe distributed OTP applications aren't the right mechanism for you in these cases.

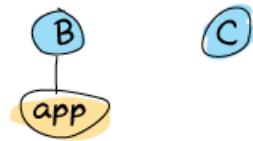
Let's imagine that we have a system with three nodes, where only the first one is running a given application:



The nodes B and C are declared to be backup nodes in case A dies, which we pretend just happened:



For a brief moment, there's nothing running. After a while, B realizes this and decides to take over the application:



That's a failover. Then, if B dies, the application gets restarted on C:



Another failover and all is well and good. Now, suppose that A comes back up. C is now running the app happily, but A is the node we defined to be the main one. This is when a takeover occurs: the app is willingly shut down on C and restarted on A:



And so on for all other failures.

One obvious problem you can see is how terminating applications all the time like that is likely to be losing important state. Sadly, that's your problem. You'll have to think of places where to put and share all that vital state away before things break down. The OTP mechanism for distributed applications makes no special case for that.

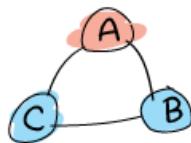
Anyway, let's move on to see how we could practically make things work.

The Magic 8-Ball

A magic 8-ball is a simple toy that you shake randomly in order to get divine and helpful answers. You ask questions like "Will my favorite sports team win

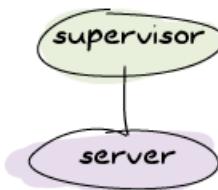
the game tonight?" and the ball you shake replies something like "Without a doubt"; you can then safely bet your house's value on the final score. Other questions like "Should I make careful investments for the future" could return "That is unlikely" or "I'm not sure". The magic 8-ball has been vital in the western world's political decision making in the last decades and it is only normal we use it as an example for fault-tolerance.

Our implementation won't make use of real-life switching mechanisms used to automatically find servers such as DNS round-robs or load balancers. We'll rather stay within pure Erlang and have three nodes (denoted below as A, B, and c) part of a distributed OTP application. The A node will represent the main node running the magic 8-ball server, and the B and c nodes will be the backup nodes:



Whenever A fails, the 8-ball application should be restarted on either B or c, and both nodes will still be able to use it transparently.

Before setting things up for distributed OTP applications, we'll first build the application itself. It's going to be mind bogglingly naive in its design:



And in total we'll have 3 modules: the supervisor, the server, and the application callback module to start things. The supervisor will be rather trivial. We'll call it m8ball_sup (as in *Magic 8 Ball Supervisor*) and we'll put it in the `src/` directory of a standard OTP application:

```
-module(m8ball_sup).  
-behaviour(supervisor).
```

```

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

start_link() ->
    supervisor:start_link({global,?MODULE}, ?MODULE, []).

init([]) ->
    {ok, [{one_for_one, 1, 10},
          [{m8ball,
            {m8ball_server, start_link, []},
            permanent,
            5000,
            worker,
            [m8ball_server]
          }]}}.

```

This is a supervisor that will start a single server (`m8ball_server`), a permanent worker process. It's allowed one failure every 10 seconds.

The magic 8-ball server will be a little bit more complex. We'll build it as a `gen_server` with the following interface:

```

-module(m8ball_server).
-behaviour(gen_server).
-export([start_link/0, stop/0, ask/1]).
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
        code_change/3, terminate/2]).


%%%%%%%
%%% INTERFACE %%%
%%%%%%%
start_link() ->
    gen_server:start_link({global, ?MODULE}, ?MODULE, [], []).

stop() ->
    gen_server:call({global, ?MODULE}, stop).

ask(_Question) -> % the question doesn't matter!
    gen_server:call({global, ?MODULE}, question).

```

Notice how the server is started using `{global, ?MODULE}` as a name and how it's accessed with the same tuple for each call. That's the `global` module we've seen in the last chapter, applied to behaviours.

Next come the callbacks, the real implementation. Before I show how we build it, I'll mention how I want it to work. The magic 8-ball should randomly pick one of many possible replies from some configuration file. I want a configuration file because it should be easy to add or remove answers as we wish.

First of all, if we want to do things randomly, we'll need to set up some randomness as part of our init function:

```
%%%%%
%%% CALLBACKS %%%
%%%%%
init([]) ->
    <<A:32, B:32, C:32>> = crypto:rand_bytes(12),
    random:seed(A,B,C),
    {ok, []}.
```

We've seen that pattern before in the [Sockets chapter](#): we're using 12 random bytes to set up the initial random seed to be used with the random:uniform/1 function.

The next step is to read the answers from the configuration file and pick one. If you recall the [OTP application chapter](#), the easiest way to set up some configuration is to use the app file to do it (in the env tuple). Here's how we're gonna do this:

```
handle_call(question, _From, State) ->
    {ok, Answers} = application:get_env(m8ball, answers),
    Answer = element(random:uniform(tuple_size(Answers)), Answers),
    {reply, Answer, State};
handle_call(stop, _From, State) ->
    {stop, normal, ok, State};
handle_call(_Call, _From, State) ->
    {noreply, State}.
```

The first clause shows what we want to do. I expect to have a tuple with all the possible answers within the answers value of the env tuple. Why a tuple? Simply because accessing elements of a tuple is a constant time operation while obtaining it from a list is linear (and thus takes longer on larger lists). We then send the answer back.

Note: the server reads the answers with `application:get_env(m8ball, answers)` on each question asked. If you were to set new answers with a call like `application:set_env(m8ball, answers, {"yes","no","maybe"})`, the three answers would instantly be the possible choices for future calls.

Reading them once at startup should be somewhat more efficient in the long run, but it will mean that the only way to update the possible answers is to restart the application.

You should have noticed by now that we don't actually care about the question asked – it's not even passed to the server. Because we're returning random answers, it is entirely useless to copy it from process to process. We're just saving work by ignoring it entirely. We still leave the answer there because it will make the final interface feel more natural. We could also trick our magic 8-ball to always return the same answer for the same question if we felt like it, but we won't bother with that for now.

The rest of the module is pretty much the same as usual for a generic `gen_server` doing nothing:

```
handle_cast(_Cast, State) ->
    {noreply, State}.

handle_info(_Info, State) ->
    {noreply, State}.

code_change(_OldVsn, State, _Extra) ->
    {ok, State}.

terminate(_Reason, _State) ->
    ok.
```

Now we can get to the more serious stuff, namely the application file and the callback module. We'll begin with the latter, `m8ball.erl`:

```
-module(m8ball).
-behaviour(application).
-export([start/2, stop/1]).
-export([ask/1]).
```

```
%%%%%%%%CALLBACKS%%%%%
%%% CALLBACKS %%%
%%%%%%%CALLBACKS%%%%%
```

```
start(normal, []) ->
    m8ball_sup:start_link().
```

```
stop(_State) ->
    ok.
```

```
%%%%%%INTERFACE%%%%%
%%% INTERFACE %%%
%%%%%INTERFACE%%%%%
ask(Question) ->
    m8ball_server:ask(Question).
```

That was easy. Here's the associated .app file, m8ball.app:

```
{application, m8ball,
  [{vsn, "1.0.0"},
   {description, "Answer vital questions"},
   {modules, [m8ball, m8ball_sup, m8ball_server]},
   {applications, [stdlib, kernel, crypto]},
   {registered, [m8ball, m8ball_sup, m8ball_server]},
   {mod, {m8ball, []}},
   {env, [
     {answers, {<<"Yes">>, <<"No">>, <<"Doubtful">>,
               <<"I don't like your tone">>, <<"Of course">>,
               <<"Of course not">>, <<"*backs away slowly and runs away*>>}}]}
  ]}.
]}.
```

We depend on stdlib and kernel, like all OTP applications, and also on crypto for our random seeds in the server. Note how the answers are all in a tuple: that matches the tuples required in the server. In this case, the answers are all binaries, but the string format doesn't really matter — a list would work as well.

Making the Application Distributed

So far, everything was like a perfectly normal OTP application. We have very few changes to add to our files to make it work for a distributed OTP

application; in fact, only one function clause to add, back in the `m8ball.erl` module:

```
%%%%%%CALLBACKS%%%%%
%%% CALLBACKS %%%
%%%%%CALLBACKS%%%%%

start(normal, []) ->
    m8ball_sup:start_link();
start({takeover, _OtherNode}, []) ->
    m8ball_sup:start_link().
```

The `{takeover, OtherNode}` argument is passed to `start/2` when a more important node takes over a backup node. In the case of the magic 8-ball app, it doesn't really change anything and we can just start the supervisor all the same.

Recompile your code and it's pretty much ready. But hold on, how do we define what nodes are the main ones and which ones are backups? The answer is in configuration files. Because we want a system with three nodes (`a`, `b`, and `c`), we'll need three configuration files (I named them `a.config`, `b.config`, and `c.config`, then put them all in `config/` inside the application directory):

```
[{kernel,
 [{distributed, [{m8ball,
      5000,
      [a@ferdmbp, {b@ferdmbp, c@ferdmbp}]}]}},
 {sync_nodes_mandatory, [b@ferdmbp, c@ferdmbp]},
 {sync_nodes_timeout, 30000}
 ]}].
```

```
[{kernel,
 [{distributed, [{m8ball,
      5000,
      [a@ferdmbp, {b@ferdmbp, c@ferdmbp}]}]}},
 {sync_nodes_mandatory, [a@ferdmbp, c@ferdmbp]},
 {sync_nodes_timeout, 30000}
 ]}].
```

```
[{kernel,
 [{distributed, [{m8ball,
```

```

    5000,
    [a@ferdmbp, {b@ferdmbp, c@ferdmbp}]}]},  

{sync_nodes_mandatory, [a@ferdmbp, b@ferdmbp]},  

{sync_nodes_timeout, 30000}  

]}].

```

Don't forget to rename the nodes to fit your own host. Otherwise the general structure is always the same:

```

[{:kernel,  

 [{distributed, [{AppName,  

      TimeOutBeforeRestart,  

      NodeList}]}],  

 {sync_nodes_mandatory, NecessaryNodes},  

 {sync_nodes_optional, OptionalNodes},  

 {sync_nodes_timeout, MaxTime}  

]}].

```

The *NodeList* value can usually take a form like [A, B, C, D] for A to be the main one, B being the first backup, and C being the next one, and so on. Another syntax is possible, giving a list of like [A, {B, C}, D], so A is still the main node, B and C are equal secondary backups, then the other ones, etc.



The `sync_nodes_mandatory` tuple will work in conjunction with `sync_nodes_timeout`. When you start a distributed virtual machine with values set for this, it will stay locked up until all the mandatory nodes are also up and locked. Then they get synchronized and things start going. If it takes more than *MaxTime* to get all the nodes up, then they will all crash before starting.

There are way more options available, and I recommend looking into the kernel application documentation if you want to know more about them.

We'll try things with the m8ball application now. If you're not sure 30 seconds is enough to boot all three VMs, you can increase the sync_nodes_timeout as you wish. Then, start three VMs:

```
$ erl -sname a -config config/a -pa ebin/  
$ erl -sname b -config config/b -pa ebin/  
$ erl -sname c -config config/c -pa ebin/
```

As you start the third VM, they should all unlock at once. Go into each of the three virtual machines, and turn by turn, start both crypto and m8ball with application:start(AppName).

You should then be able to call the magic 8-ball from any of the connected nodes:

```
(a@ferdmbp)3> m8ball:ask("If I crash, will I have a second life?").  
<<"I don't like your tone">>  
(a@ferdmbp)4> m8ball:ask("If I crash, will I have a second life, please?").  
<<"Of Course">>  
  
(c@ferdmbp)3> m8ball:ask("Am I ever gonna be good at Erlang?").  
<<"Doubtful">>
```

How motivational. To see how things are, call application:which_applications() on all nodes. Only node a should be running it:

```
(b@ferdmbp)3> application:which_applications().  
[{crypto,"CRYPTO version 2","2.1"},  
 {stdlib,"ERTS CXC 138 10","1.18"},  
 {kernel,"ERTS CXC 138 10","2.15"}]  
  
(a@ferdmbp)5> application:which_applications().  
[{m8ball,"Answer vital questions","1.0.0"},  
 {crypto,"CRYPTO version 2","2.1"},  
 {stdlib,"ERTS CXC 138 10","1.18"},  
 {kernel,"ERTS CXC 138 10","2.15"}]
```

The `c` node should show the same thing as the `b` node in that case. Now if you kill the `a` node (just ungracefully close the window that holds the Erlang shell), the application should obviously no longer be running there. Let's see where it is instead:

```
(c@ferdmbp)4> application:which_applications().  
[{crypto,"CRYPTO version 2","2.1"},  
 {stdlib,"ERTS CXC 138 10","1.18"},  
 {kernel,"ERTS CXC 138 10","2.15"}]  
(c@ferdmbp)5> m8ball:ask("where are you?!").  
<<"I don't like your tone">>
```

That's expected, as `b` is higher in the priorities. After 5 seconds (we set the timeout to 5000 milliseconds), `b` should be showing the application as running:

```
(b@ferdmbp)4> application:which_applications().  
[{m8ball,"Answer vital questions","1.0.0"},  
 {crypto,"CRYPTO version 2","2.1"},  
 {stdlib,"ERTS CXC 138 10","1.18"},  
 {kernel,"ERTS CXC 138 10","2.15"}]
```

It runs fine, still. Now kill `b` in the same barbaric manner that you used to get rid of `a`, and `c` should be running the application after 5 seconds:

```
(c@ferdmbp)6> application:which_applications().  
[{m8ball,"Answer vital questions","1.0.0"},  
 {crypto,"CRYPTO version 2","2.1"},  
 {stdlib,"ERTS CXC 138 10","1.18"},  
 {kernel,"ERTS CXC 138 10","2.15"}]
```

If you restart the node `a` with the same command we had before, it will hang. The config file specifies we need `b` back for `a` to work. If you can't expect nodes to all be up that way, you'll need to make maybe `b` or `c` optional, for example. So if we start both `a` and `b`, then the application should automatically come back, right?

```
(a@ferdmbp)4> application:which_applications().  
[{crypto,"CRYPTO version 2","2.1"},  
 {stdlib,"ERTS CXC 138 10","1.18"},  
 {kernel,"ERTS CXC 138 10","2.15"}]
```

```
(a@ferdmbp)5> m8ball:ask("is the app gonna move here?").  
<<"Of course not">>
```

Aw, shucks. The thing is, for the mechanism to work, the application needs to be started as part of the boot procedure of the node. You could, for instance, start a that way for things to work:

```
erl -sname a -config config/a -pa ebin -eval 'application:start(crypto), application:start(m8ball)'  
...  
(a@ferdmbp)1> application:which_applications().  
[{m8ball,"Answer vital questions","1.0.0"},  
 {crypto,"CRYPTO version 2","2.1"},  
 {stdlib,"ERTS CXC 138 10","1.18"},  
 {kernel,"ERTS CXC 138 10","2.15"}]
```

And from c's side:

```
=INFO REPORT==== 8-Jan-2012:19:24:27 ====  
    application: m8ball  
    exited: stopped  
    type: temporary
```

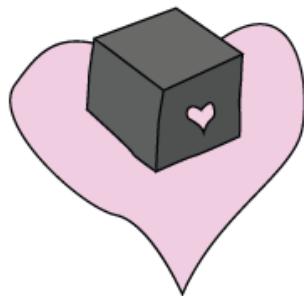
That's because the -eval option gets evaluated as part of the boot procedure of the VM. Obviously, a cleaner way to do it would be to use releases to set things up right, but the example would be pretty cumbersome if it had to combine everything we had seen before.

Just remember that in general, distributed OTP applications work best when working with releases that ensure that all the relevant parts of the system are in place.

As I mentioned earlier, in the case of many applications (the magic 8-ball included), it's sometimes simpler to just have many instances running at once and synchronizing data rather than forcing an application to run only at a single place. It's also simpler to scale it once that design has been picked. If you need some failover/takeover mechanism, distributed OTP applications might be just what you need.

Common Test for Uncommon Tests

A few chapters ago, we've seen how to use EUnit to do unit and module testing, and even some concurrent testing. At that point, EUnit started to show its limits. Complex setups and longer tests that needed to interact between each other became problematic. Plus, there was nothing in there to handle our new knowledge of distributed Erlang and all of its power. Fortunately, there's another test framework that exists, this one more appropriate to the heavy lifting we now want to do.



What is Common Test

As programmers, we enjoy treating our programs as black boxes. Many of us would define the core principle behind a good abstraction as being able to replace whatever it is we've written by an anonymous black box. You put something in the box, you get something out of it. You don't care how it works on the inside, as long as you get what you want.

In the testing world, this has an important connection to how we like to test systems. When we were working with EUnit, we've seen how to treat a module as a *black box*: you only test the exported functions and none of the ones inside, which are not exported. I've also given examples on testing items as a *white box*, like in the case of the process quest player module's tests, where we looked at the innards of the module to make its testing simpler. This was necessary because the interaction of all the

moving parts inside the box made testing it from the outside very complex.

That was for modules and functions. What if we zoom out a bit? Let's fiddle with our scope in order to see the broader picture. What if what we want to test is a library? What if it's an application? Even broader, what if it's a complete system? Then what we need is a tool that's more adept at doing something called *system testing*.

EUnit is a pretty good tool for white box testing at a module level. It's a decent tool to test libraries and OTP applications. It's possible to do system testing and black box testing, but it's not optimal.

Common Test, however, is pretty damn good at system testing. It's decent for testing libraries and OTP applications, and it's possible, but not optimal, to use it to test individual modules. So the smaller what you test is, the more appropriate (and flexible, and fun) EUnit will be. The larger your test is, the more appropriate (and flexible, and, uh, somewhat fun) Common Test will be.

You might have heard of Common Test before and tried to understand it from the documentation given with Erlang/OTP. Then you likely gave up real quick. Don't worry. The problem is that Common Test is very powerful and has an accordingly long user guide, and that at the time of this writing, most of its documentation appears to be coming from internal documentation from the days when it was used only within the walls of Ericsson. In fact, its documentation is more of a reference manual for people who already understand it than a tutorial.

In order to properly learn Common Test, we'll have to start from the simplest parts of it and slowly grow our way to system tests.

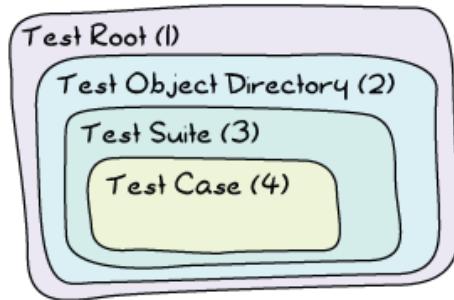
Common Test Cases

Before even getting started, I have to give you a little overview of how Common Test organises its things. First of all, because Common Test is

appropriate for system testing, it will assume two things:

1. We will need data to instantiate our stuff
2. We will need a place to store all that side-effecty stuff we do, because we're messy people.

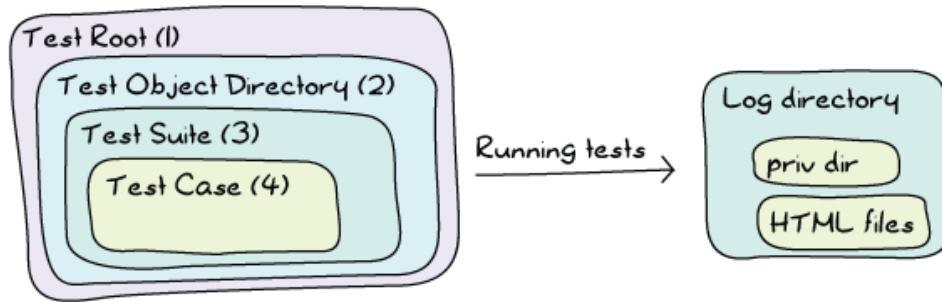
Because of this, Common Test will regularly be organized as follows:



The test case is the simplest one. It's a bit of code that either fails or succeeds. If the case crashes, the test is unsuccessful (how surprising). Otherwise, the test case is thought to be successful. In Common Test, test cases are single functions. All these functions live in a test suite (3), a module that takes care of regrouping related test cases together. Each test suite will then live in a directory, the Test Object Directory (2). The test root (1) is a directory that contains many test object directories, but due to the nature of OTP applications often being developed individually, many Erlang programmers tend to omit that layer.

In any case, now that we understand that organisation, we can go back to our two assumptions (we need to instantiate stuff, and then mess stuff up). Each test suite is a module that ends with _SUITE. If I were to test the magic 8-ball application from last chapter, I might thus call my suite m8ball_SUITE. Related to that one is a directory called the *data directory*. Each suite is allowed to have one such directory, usually named Module_SUITE_data/. In the case of the magic 8-ball app, it would have been m8ball_SUITE_data/. That directory contains anything you want.

What about the side-effects? Well because we might run tests many times, Common Test develops its structure a bit more:



Whenever you run the tests, Common Test will find some place to log stuff (usually the current directory, but we'll see how to configure it later). When doing so, it will create a unique directory where you can store your data. That directory (*Priv Dir* above), along with the data directory, will be passed as part of some initial state to each of your tests. You're then free to write whatever you want in that private directory, and then inspect it later, without running the risk of overwriting something important or the results of former test runs.

Enough with this architectural material; we're ready to write our first simple test suite. Create a directory named `ct/` (or whatever you'd like, this is hopefully a free country, after all). That directory will be our test root. Inside of it, we can then make a directory named `demo/` for the simpler tests we'll use as examples. This will be our test object directory.

Inside the test object directory, we'll begin with a module named `basic_SUITE.erl`, to see the most basic stuff possible. You can omit creating the `basic_SUITE_data/` directory — we won't need it for this run. Common Test won't complain.

Here's what the module looks like:

```
-module(basic_SUITE).
-include_lib("common_test/include/ct.hrl").
-export([all/0]).
-export([test1/1, test2/1]).
```

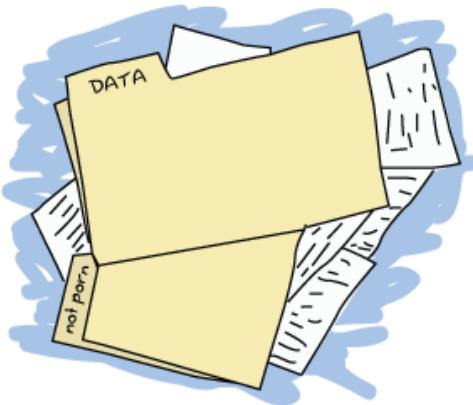
```
all() -> [test1,test2].
```

```
test1(_Config) ->  
    1 = 1.
```

```
test2(_Config) ->  
    A = 0,  
    1/A.
```

Let's study it step by step. First of all, we've got to include the file "common_test/include/ct.hrl". That file gives a few useful macros, and even though `basic_SUITE` doesn't use them, it's usually a good habit of including that file.

Then we have the function `all/0`. That function returns a list of test cases. It's basically what tells Common Test "hey, I want to run these test cases!". EUnit would do it based on the name (`*_test()` or `*_test_()`); Common Test does it with an explicit function call.



What about these `_Config` variables? They're unused for now, but for your own personal knowledge, they contain the initial state your test cases will require. That state is literally a proplist, and it initially contains two values, `data_dir` and `priv_dir`, the two directories we have for our static data and the one where we can mess around.

We can run the tests either from the command line or from an Erlang shell. If you use the command line, you can call `$ ct_run -suite Name_SUITE`. In Erlang/OTP versions before R15 (released around December 2011), the default command was `run_test` instead of `ct_run` (although some systems had both already). The name was changed with the objective of minimizing the risk of name clashes with other applications by moving to a slightly less generic name. Running it, we find:

```
ct_run -suite basic_SUITE
...
Common Test: Running make in test directories...
Recompile: basic_SUITE
...
Testing ct.demo.basic_SUITE: Starting test, 2 test cases
```

```
-----  
basic_SUITE:test2 failed on line 13
```

```
Reason: badarith  
-----
```

```
Testing ct.demo.basic_SUITE: *** FAILED *** test case 2 of 2
Testing ct.demo.basic_SUITE: TEST COMPLETE, 1 ok, 1 failed of 2 test cases
```

```
Updating /Users/ferd/code/self/learn-you-some-erlang/ct/demo/index.html... done
Updating /Users/ferd/code/self/learn-you-some-erlang/ct/demo/all_runs.html... done
```

And we find that one of our two test cases fails. We also see that we apparently inherited a bunch of HTML files. Before looking to know what this is about, let's see how to run the tests from the Erlang shell:

```
$ erl
...
1> ct:run_test([{suite, basic_SUITE}]).  

...
Testing ct.demo.basic_SUITE: Starting test, 2 test cases

-----  
basic_SUITE:test2 failed on line 13
Reason: badarith
```

...

```
Updating /Users/ferd/code/self/learn-you-some-erlang/ct/demo/index.html... done
Updating /Users/ferd/code/self/learn-you-some-erlang/ct/demo/all_runs.html... done
ok
```

I've removed a bit of the output above, but it gives exactly the same result as the command line version. Let's see what's going on with these HTML files:

```
$ ls
all_runs.html
basic_SUITE.beam
basic_SUITE.erl
ct_default.css
ct_run.NodeName.YYYY-MM-DD_20.01.25/
ct_run.NodeName.YYYY-MM-DD_20.05.17/
index.html
variables-NodeName
```

Oh what the hell did Common Test do to my beautiful directory? It is a shameful thing to look at. We've got two directories there. Feel free to explore them if you feel adventurous, but all the cowards like me will prefer to instead look at either the `all_runs.html` or the `index.html` files. The former will link to indexes of all iterations of the tests you ran while the latter will link to the newest runs only. Pick one, and then click around in a browser (or press around if you don't believe in mice as an input device) until you find the test suite with its two tests:

Num	Module	Case	Log	Time	Result	Comment
1	basic_SUITE	test1	$\leq \geq$	0.000s	Ok	
2	basic_SUITE	test2	$\leq \geq$	0.000s	FAILED	{basic_SUITE,test2, 13 } badarith
	TOTAL			0.139s	FAILED	1 Ok, 1 Failed of 2

You see that `test2` failed. If you click on the underlined line number, you'll see a raw copy of the module. If you instead click on the `test2` link, you'll

see a detailed log of what happened:

```
==== source code for basic_SUITE:test2/1
==== Test case started with:
basic_SUITE:test2(ConfigOpts)
==== Current directory is "Somewhere on my computer"
==== Started at 2012-01-20 20:05:17
[Test Related Output]
==== Ended at 2012-01-20 20:05:17
==== location [{basic_SUITE,test2,13},
    {test_server,ts_tc,1635},
    {test_server,run_test_case_eval1,1182},
    {test_server,run_test_case_eval1,1123}]
==== reason = bad argument in an arithmetic expression
in function basic_SUITE:test2/1 (basic_SUITE.erl, line 13)
in call from test_server:ts_tc/3 (test_server.erl, line 1635)
in call from test_server:run_test_case_eval1/6 (test_server.erl, line 1182)
in call from test_server:run_test_case_eval1/9 (test_server.erl, line 1123)
```

The log lets you know precisely what failed, and it is much more detailed than whatever we had in the Erlang shell. This is important to remember because if you're a shell user, you'll find Common Test extremely painful to use. If you're a person more prone to use GUIs anyway, then it'll be pretty fun for you.

But enough wandering around pretty HTML files, let's see how to test with some more state.

Note: if you ever feel like traveling back in time without the help of a time machine, download a version of Erlang prior to R15B and use Common Test with it. You'll be astonished to see that your browser and the logs' style brought you back into the late 1990s.

Testing With State

If you have read the EUnit chapter (and haven't skipped around), you'll remember that EUnit had these things called *fixtures*, where we'd give a test case some special instantiation (setup) and teardown code to be called before and after the case, respectively.

Common Test follows that concept. Instead of having EUnit-style fixtures, it instead relies on two functions. The first is the setup function, called `init_per_testcase/2` and the second one is the teardown function, called `end_per_testcase/2`. To see how they're used, create a new test suite called `state_SUITE` (still under the `demo/` directory), add the following code:

```
-module(state_SUITE).
-include_lib("common_test/include/ct.hrl").

-export([all/0, init_per_testcase/2, end_per_testcase/2]).
-export([ets_tests/1]).

all() -> [ets_tests].

init_per_testcase(ets_tests, Config) ->
    TabId = ets:new(account, [ordered_set, public]),
    ets:insert(TabId, {andy, 2131}),
    ets:insert(TabId, {david, 12}),
    ets:insert(TabId, {steve, 12943752}),
    [{table, TabId} | Config].

end_per_testcase(ets_tests, Config) ->
    ets:delete(?config(table, Config)).

ets_tests(Config) ->
    TabId = ?config(table, Config),
    [{david, 12}] = ets:lookup(TabId, david),
    steve = ets:last(TabId),
    true = ets:insert(TabId, {zachary, 99}),
    zachary = ets:last(TabId).
```

This is a little normal ETS test checking a few `ordered_set` concepts. What's interesting about it is the two new functions, `init_per_testcase/2` and `end_per_testcase/2`. Both functions need to be exported in order to be called. If they're exported, the functions are going to be called for *all* test cases in a module. You can separate them based on the arguments. The first one is the name of the test case (as an atom), and the second one is the *Config* proplist that you can modify.

Note: to read from *Config*, rather than using `proplists:get_value/2`, the Common test include file has a `?config(Key, List)` macro that returns the value matching the given key. The macro is in fact equivalent to `proplists:get_value/2` and is documented as such, so you know you can deal with *Config* as a proplist without worrying about it ever breaking.

As an example, if I had tests a, b, and c and only wanted a setup and teardown function for the first two tests, my init function might look like this:

```
init_per_testcase(a, Config) ->
    [{some_key, 124} | Config];
init_per_testcase(b, Config) ->
    [{other_key, duck} | Config];
init_per_testcase(_, Config) ->
    %% ignore for all other cases
    Config.
```

And similarly for the `end_per_testcase/2` function.

Looking back at `state_SUITE`, you can see the test case, but what's interesting to note is how I instantiate the ETS table. I specify no heir, and yet, the tests run without a problem after the init function is done.

You'll remember that we've seen, in the [ETS chapter](#), that ETS tables are usually owned by the process that started them. In this case, we leave the table as it is. If you run the tests, you'll see the suite succeeds.

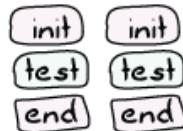
What we can infer from this is that the `init_per_testcase` and `end_per_testcase` functions run in the same process as the test case itself. You can thus safely do things like set links, start tables and whatnot without worrying about different processes breaking your things. What about errors in the test case? Fortunately, crashing in your test case won't stop Common Test from cleaning up and calling the `end_per_testcase` function, with the exception of kill exit signals.

We're now pretty much equal to EUnit with Common Test, at least in terms of flexibility, if not more. Although we haven't got all the nice assertion macros, we have fancier reports, similar fixtures, and that private directory where we can write stuff from scratch. What more do we want?

Note: if you end up feeling like outputting stuff to help you debug things or just show progress in your tests, you'll quickly find out that `io:format/1-2` prints only in the HTML logs but not the Erlang shell. If you want to do both (with free time stamps included), use the function `ct:pal/1-2`. It works like `io:format/1-2`, but prints to both the shell and logs.

Test Groups

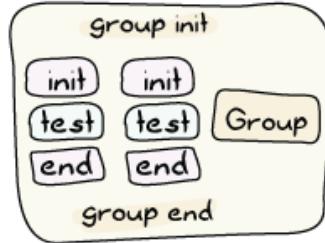
Right now, our test structure within a suite might look at best like this:



What if we have many test cases with similar needs in term of some init functions, but some different parts in them? Well, the easy way to do it is to copy/paste and modify, but this will be a real pain to maintain.

Moreover, what if what we want to do with many tests is to run them in parallel or in random order instead of one after the other? Then there's no easy way to do that based on what we've seen so far. This was pretty much the same kind of problem that could limit our use of EUnit, too.

To solve these issues, we've got something called test groups. Common Test test groups allow us to regroup some tests hierarchically. Even more, they can regroup some groups within other groups:



To make this work, we need to be able to declare the groups. The way to do it is to add a group function to declare all of them:

```
groups() -> ListOfGroups.
```

Well, there's a `groups()` function. Here's what `ListOfGroups` should be:

```
[{GroupName, GroupProperties, GroupMembers}]
```

And more in detail, here's what this could look like:

```
[{test_case_street_gang,
[], 
[simple_case, more_complex_case]}].
```

That's a tiny test case street gang. Here's a more complex one:

```
[{test_case_street_gang,
[shuffle, sequence],
[simple_case, more_complex_case,
emotionally_complex_case,
{group, name_of_another_test_group}]}].
```

That one specifies two properties, `shuffle` and `sequence`. We'll see what they mean soon. The example also shows a group including another group. This assumes that the group function might be a bit like this:

```
groups() ->
[{test_case_street_gang,
[shuffle, sequence],
[simple_case, more_complex_case, emotionally_complex_case,
{group, name_of_another_test_group}}],
{name_of_another_test_group},
```

```
[[],  
 [case1, case2, case3]]].
```

What you can do is also define the group inline within another group:

```
[{test_case_street_gang,  
 [shuffle, sequence],  
 [simple_case, more_complex_case,  
 emotionally_complex_case,  
 {name_of_another_test_group,  
 []},  
 [case1, case2, case3]}  
 ]].
```

That's getting a bit complex, right? Read them carefully, it should be simpler with time. In any case, nested groups are not a mandatory thing and you can avoid them if you find them confusing.

But wait, how do you use such a group? Well, by putting them in the `all/0` function:

```
all() -> [some_case, {group, test_case_street_gang}, other_case].
```

And that way, Common Test will be able to know whether it needs to run a test case or not.

I've quickly skipped over the group properties. We've seen `shuffle`, `sequence` and an empty list. Here's what they stand for:

empty list / no option

The test cases in the group are run one after the other. If a test fails, the others after it in the list are run.

shuffle

Runs the test in a random order. The random seed (the initialization value) used for the sequence will be printed in the HTML logs, of the form `{A,B,C}`. If a particular sequence of tests fails and you want to reproduce it, use that seed in the HTML logs and change the `shuffle`

option to instead be `{shuffle, {A,B,C}}`. That way you can reproduce random runs in their precise order if you ever need to.

parallel

The tests are run in different processes. Be careful because if you forget to export the `init_per_group` and `end_per_group` functions, Common Test will silently ignore this option.

sequence

Doesn't necessarily mean that the tests are run in order, but rather that if a test fails in the group's list, then all the other subsequent tests are skipped. This option can be combined with `shuffle` if you want any random test failing to stop the ones after.

{repeat, Times}

Repeats the group `Times` times. You could thus run all test cases in the group in parallel 9 times in a row by using the group properties `[parallel, {repeat, 9}]`. `Times` can also have the value `forever`, although '`forever`' is a bit of a lie as it can't defeat concepts such as hardware failure or heat death of the Universe (ahem).

{repeat_until_any_fail, N}

Runs all the tests until one of them fails or they have been run `N` times. `N` can also be `forever`.

{repeat_until_all_fail, N}

Same as above, but the tests may run until all cases fail.

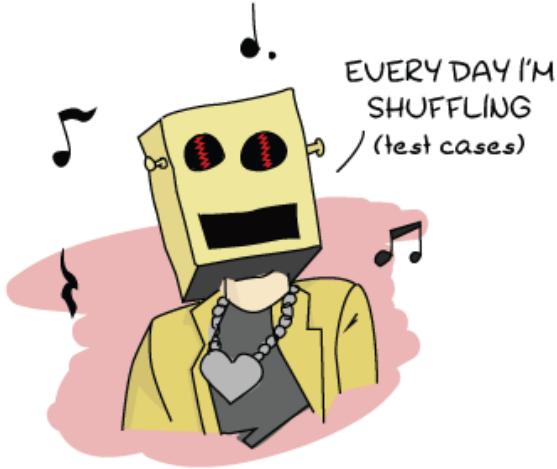
{repeat_until_any_succeed, N}

Same as before, except the tests may run until at least one case succeeds.

{repeat_until_all_succeed, N}

I think you can guess this one by yourself now, but just in case, it's the same as before except that the test cases may run until they all succeed.

Well, that's something. Honestly, that's quite a bit of content for test groups and I feel an example would be appropriate here.



The Meeting Room

To first use test groups, we'll create a meeting room booking module.

```
-module(meeting).
-export([rent_projector/1, use_chairs/1, book_room/1,
        get_all_bookings/0, start/0, stop/0]).
-record(bookings, {projector, room, chairs}).

start() ->
    Pid = spawn(fun() -> loop(#bookings{}) end),
    register(?MODULE, Pid).

stop() ->
    ?MODULE ! stop.

rent_projector(Group) ->
    ?MODULE ! {projector, Group}.

book_room(Group) ->
    ?MODULE ! {room, Group}.

use_chairs(Group) ->
    ?MODULE ! {chairs, Group}.
```

These basic functions will call a central registry process. They'll do things like allowing us to book the room, rent a projector, and put dibs on chairs. For the sake of the exercise, we're in a large organization with one hell of

a corporate structure. Because of this, there are three different people responsible for the projector, the room and the chairs, but one central registry. As such, you can't book all items at once, but must do it by sending three different messages.

To know who booked what, we can send a message to the registry in order to get all the values:

```
get_all_bookings() ->
  Ref = make_ref(),
  ?MODULE ! {self(), Ref, get_bookings},
receive
  {Ref, Reply} ->
    Reply
end.
```

The registry itself looks like this:

```
loop(B = #bookings{}) ->
receive
  stop -> ok;
  {From, Ref, get_bookings} ->
    From ! {Ref, [{room, B#bookings.room},
                  {chairs, B#bookings.chairs},
                  {projector, B#bookings.projector}]},
    loop(B);
  {room, Group} ->
    loop(B#bookings{room=Group});
  {chairs, Group} ->
    loop(B#bookings{chairs=Group});
  {projector, Group} ->
    loop(B#bookings{projector=Group})
end.
```

And that's it. To book everything for a successful meeting, we'd need to successively call:

```
1> c(meeting).
{ok,meeting}
2> meeting:start().
```

```
true
3> meeting:book_room(erlang_group).
{room,erlang_group}
4> meeting:rent_projector(erlang_group).
{projector,erlang_group}
5> meeting:use_chairs(erlang_group).
{chairs,erlang_group}
6> meeting:get_all_bookings().
[{{room,erlang_group},
 {chairs,erlang_group},
 {projector,erlang_group}}]
```

Great. This does seem wrong, though. You've possibly got this lingering feeling that things could go wrong. In many cases, if we make the three calls fast enough, we should obtain everything we want from the room without a problem. If two people do it at once and there are short pauses between the calls, it seems possible that two (or more) groups might try to rent the same equipment at once.

Oh no! Suddenly, the programmers might end up having the projector, while the board of directors has the room, and the human resources department managed to rent all chairs at once. All resources are tied up, but nobody can do anything useful!

We won't worry about fixing that problem. Instead we'll work on trying to demonstrate that it's present with a Common Test suite.

The suite, named `meeting_SUITE.erl`, will be based on the simple idea of trying to provoke a race condition that will mess up with the registration. We'll thus have three test cases, each representing a group. Carla will represent women, Mark will represent men, and a dog will represent a group of animals that somehow decided it wanted to hold a meeting with human-made tools:

```
-module(meeting_SUITE).
-include_lib("common_test/include/ct.hrl").
```

...

```
carla(_Config) ->
    meeting:book_room(women),
    timer:sleep(10),
    meeting:rent_projector(women),
    timer:sleep(10),
    meeting:use_chairs(women).
```

```
mark(_Config) ->
    meeting:rent_projector(men),
    timer:sleep(10),
    meeting:use_chairs(men),
    timer:sleep(10),
    meeting:book_room(men).
```

```
dog(_Config) ->
    meeting:rent_projector(animals),
    timer:sleep(10),
    meeting:use_chairs(animals),
    timer:sleep(10),
    meeting:book_room(animals).
```

We don't care whether these tests actually test something or not. They are just there to use the `meeting` module (which we'll see how to put in place for the tests soon) and try to generate wrong reservations.

To find out if we had a race condition or not between all of these tests, we'll make use of the `meeting:get_all_bookings()` function in a fourth and final test:

```
all_same_owner(_Config) ->
    [{_, Owner}, {_, Owner}, {_, Owner}] = meeting:get_all_bookings().
```



This one does a pattern matching on the owners of all different objects that can be booked, trying to see whether they are actually booked by the same owner. This is a desirable thing if we are looking for efficient meetings.

How do we move from having four test cases in a file to something that works? We'll need to make clever use of test groups.

First of all, because we need a race condition, we know we'll need to have a bunch of tests running in parallel. Secondly, given we have a requirement to see the problem from these race conditions, we'll need to either run `all_same_owner` many times during the whole debacle, or only after it to look with despair at the aftermath.

I chose the latter. This would give us this:

```
all() -> [{group, clients}, all_same_owner].
```

```
groups() -> [{clients,
    [parallel, {repeat, 10}],
    [carla, mark, dog]}].
```

This creates a `clients` group of tests, with the individual tests being `carla`, `mark`, and `dog`. They're going to run in parallel, 10 times each.

You see that I include the group in the `all/0` function, and then put `all_same_owner`. That's because by default, Common Test will run the tests

and groups in all/0 in the order they were declared.

But wait. We forgot to start and stop the meeting process itself. To do it, we'll need to have a way to keep a process alive for all tests, regardless of whether they're in the 'clients' group or not. The solution to this problem is to nest things one level deeper, in another group:

```
all() -> [{group, session}].
```

```
groups() -> [{session,
    [],
    [{group, clients}, all_same_owner}],
    {clients,
     [parallel, {repeat, 10}],
     [carla, mark, dog]}].
```

```
init_per_group(session, Config) ->
    meeting:start(),
    Config;
init_per_group(_, Config) ->
    Config.
```

```
end_per_group(session, _Config) ->
    meeting:stop();
end_per_group(_, _Config) ->
    ok.
```

We use the `init_per_group` and `end_per_group` functions to specify that the session group (which now runs `{group, clients}` and `all_same_owner`) will work with an active meeting. Don't forget to export the two setup and teardown functions, otherwise nothing will run in parallel.

Alright, let's run the tests and see what we get:

```
1> ct_run:run_test([{suite, meeting_SUITE}]).  
...  
Common Test: Running make in test directories...  
...  
TEST INFO: 1 test(s), 1 suite(s)
```

```
Testing ct.meeting.meeting_SUITE: Starting test (with repeated test cases)
```

```
- - - - -  
meeting_SUITE:all_same_owner failed on line 50  
Reason: {badmatch,[{room,men},{chairs,women},{projector,women}]}  
- - - - -
```

```
Testing ct.meeting.meeting_SUITE: *** FAILED *** test case 31  
Testing ct.meeting.meeting_SUITE: TEST COMPLETE, 30 ok, 1 failed of 31 test cases
```

```
...  
ok
```

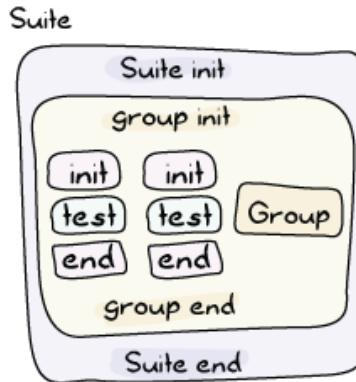
Interesting. The problem is a badmatch with three tuples with different items owned by different people. Moreover, the output tells us it's the all_same_owner test that failed. I think that's a pretty good sign that all_same_owner crashed as planned.

If you go look at the HTML log, you'll be able to see all the runs with the exact test that failed, and for what reason. Click on the test name and you'll get the right test run.

Note: one last (and very important) thing to know about before moving on from test groups is that while the init functions of test cases ran in the same process as the test case, the init functions of groups run in distinct processes from the tests. This means that whenever you initialize actors that get linked to the process that spawned them, you have to make sure to first unlink them. In the case of ETS tables, you have to define a heir to make sure it doesn't disappear. And so on for all other concepts that get attached to a process (sockets, file descriptors, etc.).

Test Suites

What can we add to our test suites that is better than nesting of groups and manipulations of how one runs things in terms of hierarchy? Not much, but we'll add another level anyway with the test suite itself:



We have two additional functions, `init_per_suite(Config)` and `end_per_suite(Config)`. These, like all the other init and end functions, aim to give more control over initialization of data and processes.

The `init_per_suite/1` and `end_per_suite/1` functions will run only once, respectively before and after all of the groups or test cases. They'll be mostly useful when dealing with general state and dependencies that will be required for all tests. This can include manually starting applications you depend on, for example.

Test Specifications

There's a thing you might have found pretty annoying if you looked at your test directory after running tests. There's a ton of files scattered around the directory for your logs. CSS files, HTML logs, directories, test run histories, etc. It would be pretty neat to have a nice way to store these files in a single directory.

Another thing is that so far we've run tests from a test suite. We've not really seen a good way to do it with many test suites at once, or even ways to only run one or two cases, or groups from a suite (or from many suites).

Of course, if I'm saying this, it's because I've got a solution for these issues. There are ways to do it both from the command line and from the Erlang shell, and you can find them in the documentation for `ct_run`.

However, instead of going into ways to manually specify everything for each time you run the tests, we'll see something called *test specifications*.



Test specifications are special files that let you detail everything about how you want to have the tests run, and they work with the Erlang shell and the command line. The test specification can be put in a file with any extension you want (although I personally fancy .spec files). The spec files will contain Erlang tuples, much like a consult file. Here's a few of the items it can have:

{include, *IncludeDirectories*}

When Common Test automatically compiles suites, this option lets you specify where it should look for include files in order to make sure they're there. The *IncludeDirectories* value has to be a string (list) or a list of strings (list of lists).

{logdir, *LoggingDirectory*}

When logging, all logs should be moved to the *LoggingDirectory*, a string. Note that the directory must exist before the tests are run, otherwise Common Test will complain.

{suites, *Directory*, *Suites*}

Finds the given suites in *Directory*. *Suites* can be an atom (some_SUITE), a list of atoms, or the atom `all` to run all the suites in a directory.

{skip_suites, *Directory*, *Suites*, *Comment*}

This subtracts a list of suites from those previously declared and skips them. The *Comment* argument is a string explaining why you decided to skip them. This comment will be put in the final HTML logs. The tables will show a yellow 'SKIPPED: Reason' where *Reason* is whatever *Comment* contained.

{groups, Directory, Suite, Groups}

This is an option to pick only a few groups from a given suite. The *Groups* variable can be a single atom (the group name) or `all` for all groups. The value can also be more complex, letting you override the group definitions inside `groups()` within the test case by picking a value like `{GroupName, [parallel]}`, which will override *GroupName*'s options for `parallel`, without needing to recompile tests.

{groups, Directory, Suite, Groups, {cases,Cases}}

Similar to the one above, but it lets you specify some test cases to include in the tests by substituting *Cases* by a single case name (an atom), a list of names, or the atom `all`.

{skip_groups, Directory, Suite, Groups, Comment}

This command was only added in R15B and documented in R15B01. It allows one to skip test groups, much like the `skip_suites` for suites.

There is no explanation as to why it wasn't there before then.

{skip_groups, Directory, Suite, Groups, {cases,Cases}, Comment}

Similar to the one above, but with specific test cases to skip on top of it. Also only available since R15B.

{cases, Directory, Suite, Cases}

Runs specific test cases from a given suite. *Cases* can be an atom, a list of atoms, or `all`.

{skip_cases, Directory, Suite, Cases, Comment}

This is similar to `skip_suites`, except we choose specific test cases to avoid with this one.

{alias, Alias, Directory}

Because it gets very annoying to write all these directory names (especially if they're full names), Common Test lets you substitute them with aliases (atoms). This is pretty useful in order to be concise.

Before showing a simple example, you should add a `logs/` directory above the `demo/` one (`ct/` in my files). Unsurprisingly, that's where our Common Test logs will be moved to. Here's what a possible test specification could look like for all our tests so far, under the imaginative name of `spec.spec`:

```

{alias, demo, "./demo/"}.
{alias, meeting, "./meeting/"}.
{logdir, "./logs/"}.

{suites, meeting, all}.
{suites, demo, all}.
{skip_cases, demo, basic_SUITE, test2, "This test fails on purpose"}.

```

This spec file declares two aliases, `demo` and `meeting`, which point to the two test directories we have. We put the logs inside `ct/logs/`, our newest directory. Then we ask to run all suites in the `meeting` directory, which, coincidentally is the `meeting_SUITE` suite. Next on the list are the two suites inside the `demo` directory. Moreover, we ask to skip `test2` from the `basic_SUITE` suite, given it contains a division by zero that we know will fail.

To run the tests, you can either use `$ ct_run -spec spec.spec` (or `run_test` for versions of Erlang before R15), or you can use the function `ct:run_test([{spec, "spec.spec"}])`. from the Erlang shell:

Common Test: Running make in test directories...

...

TEST INFO: 2 test(s), 3 suite(s)

Testing `ct.meeting`: Starting test (with repeated test cases)

```
-----  
meeting_SUITE:all_same_owner failed on line 51  
Reason: {badmatch,[{room,men},{chairs,women},{projector,women}]}  
-----
```

Testing `ct.meeting`: *** FAILED *** test case 31

Testing `ct.meeting`: TEST COMPLETE, 30 ok, 1 failed of 31 test cases

Testing `ct.demo`: Starting test, 3 test cases

Testing `ct.demo`: TEST COMPLETE, 2 ok, 0 failed, 1 skipped of 3 test cases

```
Updating /Users/ferd/code/self/learn-you-some-erlang/ct/logs/index.html... done  
Updating /Users/ferd/code/self/learn-you-some-erlang/ct/logs/all_runs.html... done
```

If you take the time to look at the logs, you'll see two directories for the different test runs. One of them will have a failure; that's the meeting that fails as expected. The other one will have one success, and one skipped case, of the form 1 (1/0). Generally, the format is TotalSkipped (IntentionallySkipped/SkippedDueToError). In this case the skip happened from the spec file, so it goes on the left. If it happened because one of the many init functions failed, then it'd be on the right.

Common Test is starting to look like a pretty decent testing framework, but it'd be pretty nice to be able to use our distributed programming knowledge and apply it.



Large Scale Testing

Common Test does support having distributed tests. Before going hog wild and writing a bunch of code, let's see what's offered. Well, there isn't *that* much. The gist of it is that Common Test lets you start tests on many different nodes, but also has ways to dynamically start these nodes and have them watch each other.

As such, the distributed features of Common Test are really useful when you have large test suites that should be run in parallel on many nodes. This is often worth it to save time or because the code will run in

production environments that are on different computers — automated tests that reflect this are desired.

When tests go distributed, Common Test requires the presence of a central node (the *CT master*) in charge of all the other ones. Everything's going to be directed from there, from starting nodes, ordering tests to be run, gathering logs, etc.

The first step to get things going that way is to expand our test specifications so they become distributed. We're going to add a few new tuples:

{node, NodeAlias, NodeName}

Much like {alias, AliasAtom, Directory} for test suites, groups, and cases, except it's used for node names. Both *NodeAlias* and *NodeName* need to be atoms. This tuple is especially useful because *NodeName* needs to be a long node name, and in some cases this can be quite long.

{init, NodeAlias, Options}

This is a more complex one. This is the option that lets you start nodes. *NodeAlias* can be a single node alias, or a list of many of them. The *Options* are those available to the *ct_slave* module:

Here are a few of the options available:

{username, UserName} and {password, Password}

Using the host part of the node given by *NodeAlias*, Common Test will try to connect to the given host over SSH (on port 22) using the user name and password and run from there.

{startup_functions, [{M,F,A}]}{}

This option defines a list of functions to be called as soon as the other node has booted.

{erl_flags, String}

This sets standard flags that we'd want to pass to the *erl* application when we start it. For example, if we wanted to start a node with *erl -*

```
env ERL_LIBS .. / -config conf_file, the option would be {erl_flags, "-env ERL_LIBS .. /  
-config config_file"}.
```

{monitor_master, true | false}

If the CT master stops running and the option is set to true, then the slave node will also be taken down. I do recommend using this option if you're spawning the remote nodes; otherwise they'll keep running in the background if the master dies. Moreover, if you run tests again, Common Test will be able to connect to these nodes, and there will be some state attached to them.

```
{boot_timeout, Seconds},  
{init_timeout, Seconds},  
{startup_timeout, Seconds}
```

These three options let you wait for different parts of the starting of a remote node. The boot timeout is about how long it takes before the node becomes pingable, with a default value of 3 seconds. The init timeout is an internal timer where the new remote node calls back the CT master to tell it it's up. By default, it lasts one second. Finally, the startup timeout tells Common Test how long to wait for the functions we defined earlier as part of the startup_functions tuple.

{kill_if_fail, true | false}

This option will react to one of the three timeouts above. If any of them are triggered, Common Test will abort the connection, skip tests, etc. but not necessarily kill the node, unless the option is set to true. Fortunately, that's the default value.

Note: all these options are provided by the `ct_slave` module. It is possible to define your own module to start slave nodes, as long as it respects the right interface.

That makes for quite a lot of options for remote nodes, but that's partially what gives Common Test its distributed power; you're able to boot nodes with pretty much as much control as what you'd get doing it by hand in the shell. Still, there are more options for distributed tests, although they're not for booting nodes:

```

{include, Nodes, IncludeDirs}
{logdir, Nodes, LogDir}
{suites, Nodes, DirectoryOrAlias, Suites}
{groups, Nodes, DirectoryOrAlias, Suite, Groups}
{groups, Nodes, DirectoryOrAlias, Suite, GroupSpec, {cases,Cases}}
{cases, Nodes, DirectoryOrAlias, Suite, Cases}
{skip_suites, Nodes, DirectoryOrAlias, Suites, Comment}
{skip_cases, Nodes, DirectoryOrAlias, Suite, Cases, Comment}

```

These are pretty much the same as what we've already seen, except that they can optionally take a node argument to add more detail. That way you can decide to run some suites on a given node, others on different nodes, etc. This could be useful when doing system testing with different nodes running different environments or parts of the system (such as databases, external applications, etc.)

As a simple way to see how this works, let's turn the previous spec.spec file into a distributed one. Copy it as dist.spec and then change it until it looks like this:

```

{node, a, 'a@ferdmbp.local'}.
{node, b, 'b@ferdmbp.local'}.

{init, [a,b], [{node_start, [{monitor_master, true}]}]}.

{alias, demo, "./demo/"}.
{alias, meeting, "./meeting/"}.

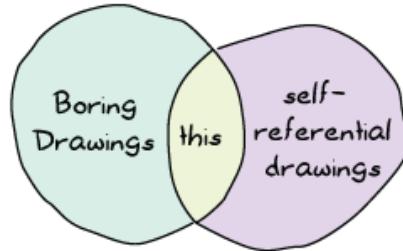
{logdir, all_nodes, "./logs/"}.
{logdir, master, "./logs/"}.

{suites, [b], meeting, all}.
{suites, [a], demo, all}.
{skip_cases, [a], demo, basic_SUITE, test2, "This test fails on purpose"}.

```

This changes it a bit. We define two slave nodes, a and b, that need to be started for the tests. They do nothing special but make sure to die if the master dies. The aliases for directories remain the same as they were.

The logdir values are interesting. We declared no node alias as all_nodes or master, but yet, here they are. The all_nodes alias stands for all non-master nodes for Common Test, while master stands for the master node itself. To truly include all nodes, both all_nodes and master are required. No clear explanation as to why these names were picked.



The reason why I put all values there is that Common Test will generate logs (and directories) for each of the slave nodes, and it will also generate a master set of logs, referring to the slave ones. I don't want any of these in directories other than `logs/`. Note that the logs for the slave nodes will be stored on each of the slave nodes individually. In that case, unless all nodes share the same filesystem, the HTML links in the master's logs won't work and you'll have to access each of the nodes to get their respective logs.

Last of all are the `suites` and `skip_cases` entries. They're pretty much the same as the previous ones, but adapted for each node. This way, you can skip some entries only on given nodes (which you know might be missing libraries or dependencies), or maybe more intensive ones where the hardware isn't up to the task.

To run distributed tests of the sort, we must start a distributed node with `-name` and use `ct_master` to run the suites:

```
$ erl -name ct
Erlang R15B (erts-5.9) [source] [64-bit] [smp:4:4] [async-threads:0] [hipe] [kernel-poll:false]

Eshell V5.9 (abort with ^G)
(ct@ferdmbp.local)1> ct_master:run("dist.spec").
== Master Logdir ==
```

```

/Users/ferd/code/self/learn-you-some-erlang/ct/logs
==== Master Logger process started ====
<0.46.0>
Node 'a@ferdmbp.local' started successfully with callback ct_slave
Node 'b@ferdmbp.local' started successfully with callback ct_slave
==== Cookie ====
'PMIYERCHJZNZGSRJPVRK'
==== Starting Tests ====
Tests starting on: ['b@ferdmbp.local','a@ferdmbp.local']
==== Test Info ====
Starting test(s) on 'b@ferdmbp.local'...
==== Test Info ====
Starting test(s) on 'a@ferdmbp.local'...
==== Test Info ====
Test(s) on node 'a@ferdmbp.local' finished.
==== Test Info ====
Test(s) on node 'b@ferdmbp.local' finished.
==== TEST RESULTS ====
a@ferdmbp.local-----finished_ok
b@ferdmbp.local-----finished_ok

==== Info ====
Updating log files
Updating /Users/ferd/code/self/learn-you-some-erlang/ct/logs/index.html... done
Updating /Users/ferd/code/self/learn-you-some-erlang/ct/logs/all_runs.html... done
Logs in /Users/ferd/code/self/learn-you-some-erlang/ct/logs refreshed!
==== Info ====
Refreshing logs in "/Users/ferd/code/self/learn-you-some-erlang/ct/logs"... ok
[{"dist.spec","ok"}]

```

There is no way to run such tests using `ct_run`. Note that CT will show all results as `ok` whether or not the tests actually succeeded. That is because `ct_master` only shows if it could contact all the nodes. The results themselves are actually stored on each individual node.

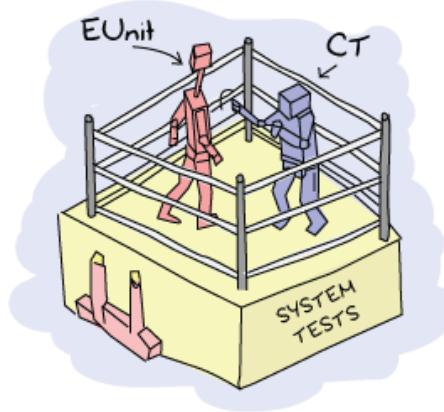
You'll also note that CT shows that it started nodes, and with what cookies it did so. If you try running tests again without first terminating the master, the following warnings are shown instead:

WARNING: Node 'a@ferdmbp.local' is alive but has node_start option

WARNING: Node 'b@ferdmbp.local' is alive but has node_start option

That's alright. It only means that Common Test is able to connect to remote nodes, but it found no use in calling our init tuple from the test specification, given the nodes are already alive. There is no need for Common Test to actually start any remote nodes it will run tests on, but I usually find it useful to do so.

That's really the gist of distributed spec files. Of course you can get into more complex cases, where you set up more complex clusters and write amazing distributed tests, but as the tests become more complex, the less confidence you can have in their ability to successfully demonstrate properties of your software, simply because tests themselves might contain more and more errors as they become convoluted.



Integrating EUnit within Common Test

Because sometimes EUnit is the best tool for the job, and sometimes Common Test is, it might be desirable for you to include one into the other.

While it's difficult to include Common Test suites within EUnit ones, the opposite is quite easy to do. The trick is that when you call `eunit:test(SomeModule)`, the function can return either `ok` when things work, or `error` in case of any failure.

This means that to integrate EUnit tests to a Common Test suite, all you need to do is have a function a bit like this:

```
run_eunit(_Config) ->  
    ok = eunit:test(TestsToRun).
```

And all your EUnit tests that can be found by the *TestsToRun* description will be run. If there's a failure, it'll appear in your Common Test logs and you'll be able to read the output to see what went wrong. It's that simple.

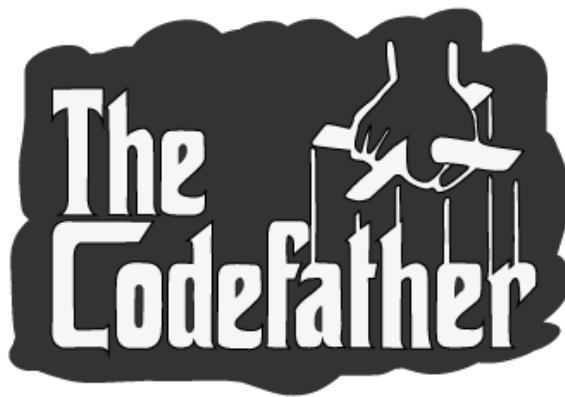
Is There More?

You bet there's more. Common Test is a very complex beast. There are ways to add configuration files for some variables, add hooks that run at many points during the test executions, use callbacks on events during the suites, modules to test over SSH, Telnet, SNMP, and FTP.

This chapter only scratched the surface, but it is enough to get you started if you want to explore in more depth. A more complete document about Common Test is the user's guide coming with Erlang/OTP. It is hard to read on its own, but understanding the material covered in this very chapter will help you figure out the documentation, without a doubt.

Mnesia And The Art of Remembering

You're the closest friend of a man with friends. Many of them. Some for a very long time, much like you. They come from all around the world, ranging from Sicily to New York. Friends pay their respects, care about you and your friend, and you both care about them back.



In exceptional circumstances, they ask for favors because you're people of power, people of trust. They're your good friends, so you oblige. However, friendship has a cost. Each favor realized is duly noted, and at some point in the future, you may or may not ask for a service back.

You always hold your promises, you're a pillar of reliability. That's why they call your friend *boss*, they call you *consigliere*, and why you're leading one of the most respected mafia families.

However, it becomes a pain to remember all your friendships, and as your areas of influence grow across the world, it is increasingly harder to keep track of what friends owe to you, and what you owe to friends.

Because you're a helpful counselor, you decide to upgrade the traditional system from notes secretly kept in various places to something using Erlang.

At first you figure using ETS and DETS tables will be perfect. However, when you're out on an overseas trip away from the boss, it becomes somewhat difficult to keep things synchronized.

You could write a complex layer on top of your ETS and DETS tables to keep everything in check. You could do that, but being human, you know you would make mistakes and write buggy software. Such mistakes are to be avoided when friendship is so important, so you look online to find how to make sure your system works right.

This is when you start reading this chapter, explaining Mnesia, an Erlang distributed database built to solve such problems.

What's Mnesia

Mnesia is a layer built on top of ETS and DETS to add a lot of functionality to these two databases. It mostly contains things many developers might end up writing on their own if they were to use them intensively. Features include the ability to write to both ETS and DETS automatically, to both have DETS' persistence and ETS' performance, or having the possibility to replicate the database to many different Erlang nodes automatically.

Another feature we've seen to be useful is *transactions*. Transactions basically mean that you're going to be able to do multiple operations on one or more tables as if the process doing them were the only one to have access to the tables. This is going to prove vital as soon as we need to have concurrent operations that mix read and writes as part of a single unit. One example would be reading in the database to see if a username is taken, and then creating the user if it's free. Without transactions, looking inside the table for the value and then registering it counts as two distinct operations that can be messing with each other – given the right timing, more than one process at a time might believe it has the right to create the unique user, which will lead to a lot

of confusion. Transactions solve this problem by allowing many operations to act as a single unit.

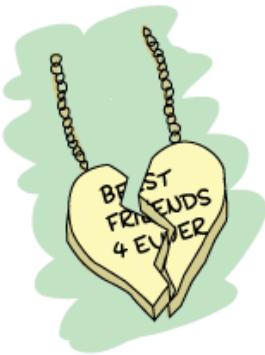
The nice thing about Mnesia is that it's pretty much the only full-featured database you'll have that will natively store and return any Erlang term out of the box (at the time of this writing). The downside of that is that it will inherit all the limitations of DETS tables in some modes, such as not being able to store more than 2GB of data for a single table on disk (this can in fact be bypassed with a feature called fragmentation.)

If we refer to the CAP theorem, Mnesia sits on the CP side, rather than the AP side, meaning that it won't do eventual consistency, will react rather badly to netsplits in some cases, but will give you strong consistency guarantees if you expect the network to be reliable (and you sometimes shouldn't).

Note that Mnesia is not meant to replace your standard SQL database, and it's also not meant to handle terabytes of data across a large number of data centers as often claimed by the giants of the NoSQL world. Mnesia is rather made for smaller amounts of data, on a limited number of nodes. While it is possible to use it on a ton of nodes, most people find that their practical limits seem to center around 10 or so. You will want to use Mnesia when you know it will run on a fixed number of nodes, have an idea of how much data it will require, and know that you will primarily need to access your data from Erlang in ways ETS and DETS would let you do it in usual circumstances.

Just how close to Erlang is it? Mnesia is centered around the idea of using a record to define a table's structure. Each table can thus store a bunch of similar records, and anything that goes in a record can thus be stored in a Mnesia table, including atoms, pids, references, and so on.

What Should the Store Store



The first step in using Mnesia is to figure out what kind of table structure we'll want for our mafia friend-tracking application (which I decided to name mafiapp). The information we might want to store related to friends will be:

- the friend's name, to know who we're talking to when we ask for a service or when we give one
- the friend's contact information, to know how to reach them. It can be anything from an e-mail address, a cell phone number, or even notes of where that person likes to hang out
- additional information such as when the person was born, their occupation, hobbies, special traits, and so on
- a unique expertise, our friend's forte. This field stands on its own because it's something we want to know explicitly. If someone's expertise or forte is in cooking and we're in dire need of a caterer, we know who to call. If we are in trouble and need to disappear for a while, maybe we'll have friends with expertises such as being able to pilot a plane, being camouflage experts, or possibly being excellent magicians. This could come in handy.

Then we have to think of the services between our friends and us. What will we want to know about them? Here's a list of a few things I can think about:

1. Who gave the service. Maybe it's you, the consigliere. Maybe it's the padrino. Maybe it's a friend of a friend, on your behalf. Maybe it's

- someone who then becomes your friend. We need to know.
2. Who received the service. Pretty much the same as the previous one, but on the receiving end.
 3. When was the service given. It's generally useful to be able to refresh someone's memory, especially when asking for a favor back.
 4. Related to the previous point, it would be nice to be able to store details regarding the services. It's much nicer (and more intimidating) to remember every tiny detail of the services we gave on top of the date.

As I mentioned in the previous section, Mnesia is based on records and tables (ETS and DETS). To be exact, you can define an Erlang record and tell Mnesia to turn its definition into a table. Basically, if we decided to have our record take the form:

```
-record(recipe, {name, ingredients=[], instructions=[]}).
```

We can then tell Mnesia to create a recipe table, which would store any number of `#recipe{}` records as table rows. I could thus have a recipe for pizza noted as:

```
#recipe{name=pizza,  
        ingredients=[sauce,tomatoes,meat,dough],  
        instructions=["order by phone"]}
```

and a recipe for soup as:

```
#recipe{name=soup,  
        ingredients=["who knows"],  
        instructions=["open unlabeled can, hope for the best"]}
```

And I could insert both of these in the recipe table, as is. I could then fetch the same exact records from the table and use them as any other one.

The primary key, the field by which it is the fastest to look things up in a table, would be the recipe name. That's because name is the first item in the record definition for `#recipe{}`. You'll also notice that in the pizza recipe, I use atoms as ingredients, and in the soup recipe, I use a string. As opposed to SQL tables, Mnesia tables *have no built-in type constraints*, as long as you respect the tuple structure of the table itself.

Anyway, back to our mafia application. How should we represent our friends and services information? Maybe as one table doing everything?

```
-record(friends, {name,  
                 contact=[],  
                 info=[],  
                 expertise,  
                 service=[]}). % {To, From, Date, Description} for services?
```

This isn't the best choice possible though. Nesting the data for services within friend-related data means that adding or modifying service-related information will require us to change friends at the same time. This might be annoying to do, especially since services imply at least two people. For each service, we would need to fetch the records for two friends and update them, even if there is no friend-specific information that needs to be modified.

A more flexible model would use one table for each kind of data we need to store:

```
-record(mafiapp_friends, {name,  
                           contact=[],  
                           info=[],  
                           expertise}).  
  
-record(mafiapp_services, {from,  
                           to,  
                           date,  
                           description}).
```

Having two tables should give us all the flexibility we need to search for information, modify it, and with little overhead. Before getting into how to handle all that precious information, we must initialize the tables.

Don't Drink Too Much Kool-Aid:

you'll notice that I prefixed both the friends and services records with mafiapp_. The reason for this is that while records are defined locally within our module, Mnesia tables are global to all the nodes that will be part of its cluster. This implies a high potential for name clashes if you're not careful. As such, it is a good idea to manually namespace your tables.

From Record to Table

Now that we know what we want to store, the next logical step is to decide how we're going to store it. Remember that Mnesia is built using ETS and DETS tables. This gives us two means of storage: on disk, or in memory. We have to pick a strategy! Here are the options:

`ram_copies`

This option makes it so all data is stored exclusively in ETS, so memory only. Memory should be limited to a theoretical 4GB (and practically around 3GB) for virtual machines compiled on 32 bits, but this limit is pushed further away on 64 bits virtual machines, assuming there is more than 4GB of memory available.

`disc_only_copies`

This option means that the data is stored only in DETS. Disc only, and as such the storage is limited to DETS' 2GB limit.

`disc_copies`

This option means that the data is stored both in ETS and on disk, so both memory and the hard disk. `disc_copies` tables are *not* limited by DETS limits, as Mnesia uses a complex system of transaction logs and checkpoints that allow to create a disk-based backup of the table in memory.

For our current application, we will go with `disc_copies`. The reason for this is that we at least need the persistency to disk. The relationships we built with our friends need to be long-lasting, and as such it makes sense to be able to store things persistently. It would be quite annoying to wake up after a power failure, only to find out you've lost all the friendships you worked so hard for. Why just not use `disc_only_copies`, you might ask? Well, having copies in memory is usually nice when we want to do more somewhat complex queries and search, given they can be done without needing to access the disc, which is often the slowest part of any computer memory access, especially if they're hard discs.

There's another hurdle on our path to filling the database with our precious data. Because of how ETS and DETS work, we need to define a table type. The types available bear the same definition as their ETS and DETS counterparts. The options are `set`, `bag`, and `ordered_set`. `ordered_set` specifically is not supported for `disc_only_copies` tables. If you don't remember what these types do, I recommend you look them up in the [ETS chapter](#).

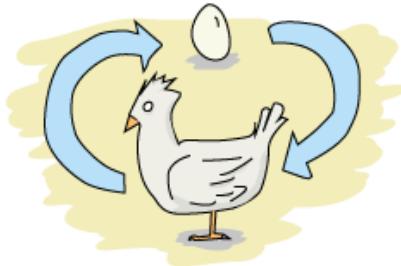
Note: Tables of type `duplicate_bag` are not available for any of the storage types. There is no obvious explanation as to why that is.

The good news is that we're pretty much done deciding how we're going to store things. The bad news is that there are still more things to understand about Mnesia before truly getting started.

Of Schemas and Mnesia

Although Mnesia can work fine on isolated nodes, it does support distribution and replication to many nodes. To know how to store tables on disk, how to load them, and what other nodes they should be synchronized with, Mnesia needs to have something called a *schema*, holding all that information. By default, Mnesia creates a schema directly in memory when it's created. It works fine for tables that need

to live in RAM only, but when your schema needs to survive across many VM restarts, on all the nodes part of the Mnesia cluster, things get a bit more complex.



Mnesia depends on the schema, but Mnesia should also create the schema. This creates a weird situation where the schema needs to be created by Mnesia without running Mnesia first! It's rather simple to solve as a problem in practice. We just have to call the function `mnesia:create_schema(ListOfNodes)` *before* starting Mnesia. It will create a bunch of files on each node, storing all the table information required. You don't need to be connected to the other nodes when calling it, but they need to be running; the function will set the connections up and get everything working for you.

By default, the schema will be created in the current working directory, wherever the Erlang node is running. To change this, the Mnesia application has a `dir` variable that can be set to pick where the schema will be stored. You can thus start your node as `erl -name SomeName -mnesia dir where/to/store/the/db` or set it dynamically with `application:set_env(mnesia, dir, "where/to/store/the/db")`.

Note: Schemas may fail to be created for the following reasons: one already exists, Mnesia is running on one of the nodes the schema should be on, you can't write to the directory Mnesia wants to write to, and so on.

Once the schema has been created, we can start Mnesia and begin creating tables. The function `mnesia:create_table/2` is what we need to use.

It takes two arguments: the table name and a list of options, some of which are described below.

{attributes, List}

This is a list of all the items in a table. By default it takes the form [key, value], meaning you would need a record of the form - record(TableName, {key,value}). to work. Pretty much everyone cheats a little bit and uses a special construct (a compiler-supported macro, in fact) that extracts the element names from a record. The construct looks like a function call. To do it with our friends record, we would pass it as {attributes, record_info(fields, mafiapp_friends)}.

{disc_copies, NodeList},

{disc_only_copies, NodeList},

{ram_copies, NodeList}

This is where you specify how to store the tables, as explained in [From Record to Table](#). Note that you can have many of these options present at once. As an example, I could define a table X to be stored on disk and RAM on my master node, only in RAM on all of the slaves, and only on disk on a dedicated backup node by using all three of the options.

{index, ListOfIntegers}

Mnesia tables let you have *indexes* on top of the basic ETS and DETS functionality. This is useful in cases where you are planning to build searches on record fields other than the primary key. As an example, our friends table will need an index for the expertise field. We can declare such an index as {index, [#mafiapp_friends.expertise]}. In general, and this is true for many, many databases, you want to build indexes only on fields that are not too similar between most entries. On a table with hundreds of thousands of entries, if your index at best splits your table in two groups to sort through, indexing will take a lot of place for very little benefit. An index that would split the same table in N groups of ten or less elements, as an example, would be more useful for the resources it uses. Note

that you do not need to put an index on the first field of the record, as this is done for you by default.

{record_name, Atom}

This is useful if you want to have a table that has a different name than the one your record uses. However, doing so then forces you to use different functions to operate on the table than those commonly used by everyone. I wouldn't recommend using this option, unless you really know you want to.

{type, Type}

Type is either set, ordered_set or bag tables. This is the same as what I have explained earlier in [From Record to Table](#).

{local_content, true | false}

By default, all Mnesia tables have this option set to false. You will want to leave it that way if you want the tables and their data replicated on all nodes part of the schema (and those specified in the disc_copies, disc_only_copies and ram_copies options). Setting this option to true will create all the tables on all the nodes, but the content will be the local content only; nothing will be shared. In this case, Mnesia becomes an engine to initialize similar empty tables on many nodes.

To make things short, this is the sequence of events that can happen when setting up your Mnesia schema and tables:

- Starting Mnesia for the first time creates a schema in memory, which is good for ram_copies. Other kinds of tables won't work with it.
- If you create a schema manually before starting Mnesia (or after stopping it), you will be able to create tables that sit on disk.
- Start Mnesia, and you can then start creating tables. Tables can't be created while Mnesia is not running

Note: there is a third way to do things. Whenever you have a Mnesia node running and tables created that you would want to port to disk,

the function `mnesia:change_table_copy_type(Table, Node, NewType)` can be called to move a table to disk.

More particularly, if you forgot to create the schema on disk, by calling `mnesia:change_table_copy_type(schema, node(), disc_copies)`, you'll be taking your RAM schema and turning it to a disk schema.

We now have a vague idea of how to create tables and schemas. This might be enough for us to get started.

Creating Tables for Real

We'll handle creating the application and its tables with some weak TDD-style programming, using Common Test. Now you might dislike the idea of TDD, but stay with me, we'll do it in a relaxed manner, just as a way to guide our design more than anything else. None of that 'run tests to make sure they fail' business (although you can feel free to do it if you want). That we have tests in the end will just be a nice side-effect, not an end in itself. We'll mostly care about defining the interface of how mafiapp should behave and look like, without doing it all from the Erlang shell. The tests won't even be distributed, but it will still be a decent opportunity to get some practical use out of Common Test while learning Mnesia at the same time.

For this, we should start a directory named mafiapp-1.0.0 following the standard OTP structure:

```
ebin/  
logs/  
src/  
test/
```

We'll start by figuring out how we want to install the database. Because there is a need for a schema and initializing tables the first time around, we'll need to set up all the tests with an `install` function that will

ideally install things in Common Test's priv_dir directory. Let's begin with a basic test suite, mafiapp_SUITE, stored under the test/ directory:

```
-module(mafiapp_SUITE).  
-include_lib("common_test/include/ct.hrl").  
-export([init_per_suite/1, end_per_suite/1,  
        all/0]).  
all() -> [].
```

```
init_per_suite(Config) ->  
    Priv = ?config(priv_dir, Config),  
    application:set_env(mnesia, dir, Priv),  
    mafiapp:install([node()]),  
    application:start(mnesia),  
    application:start(mafiapp),  
    Config.  
  
end_per_suite(_Config) ->  
    application:stop(mnesia),  
    ok.
```

This test suite has no test yet, but it gives us our first specification of how things should be done. We first pick where to put the Mnesia schema and database files by setting the `dir` variable to the value of `priv_dir`. This will put each instance of the schema and database in a private directory generated with Common Test, guaranteeing us not to have problems and clashes from earlier test runs. You can also see that I decided to name the install function `install` and to give it a list of nodes to install to. Such a list is generally a better way to do things than hard coding it within the `install` function, as it is more flexible. Once this is done, Mnesia and mafiapp should be started.

We can now get into `src/mafiapp.erl` and start figuring out how the `install` function should work. First of all, we'll need to take the record definitions we had earlier and bring them back in:

```
-module(mafiapp).  
-export([install/1]).
```

```

-record(mafiapp_friends, {name,
    contact=[],
    info=[],
    expertise}).

-record(mafiapp_services, {from,
    to,
    date,
    description}).

```

This looks good enough. Here's the `install/1` function:

```

install(Nodes) ->
    ok = mnesia:create_schema(Nodes),
    application:start(mnesia),
    mnesia:create_table(mafiapp_friends,
        [{attributes, record_info(fields, mafiapp_friends)},
         {index, [#mafiapp_friends.expertise]},
         {disc_copies, Nodes}]),
    mnesia:create_table(mafiapp_services,
        [{attributes, record_info(fields, mafiapp_services)},
         {index, [#mafiapp_services.to]},
         {disc_copies, Nodes},
         {type, bag}]),
    application:stop(mnesia).

```

First, we create the schema on the nodes specified in the `Nodes` list. Then, we start Mnesia, which is a necessary step in order to create tables. We create the two tables, named after the records `#mafiapp_friends{}` and `#mafiapp_services{}`. There's an index on the `expertise` because we do expect to search friends by expertise in case of need, as mentioned earlier.



You'll also see that the services table is of type bag. This is because it's possible to have multiple services with the same senders and receivers. Using a set table, we could only deal with unique senders, but bag tables handle this fine. Then you'll notice there's an index on the to field of the table. That's because we expect to look services up either by who received them or who gave them, and indexes allow us to make any field faster to search.

Last thing to note is that I stop Mnesia after creating the tables. This is just to fit whatever I wrote in the test in terms of behaviour. What was in the test is how I expect to use the code, so I'd better make the code fit that idea. There is nothing wrong with just leaving Mnesia running after the install, though.

Now, if we had successful test cases in our Common Test suite, the initialization phase would succeed with this install function. However, trying it with many nodes would bring failure messages to our Erlang shells. Any idea why? Here's what it would look like:

Node A	Node B
-----	-----
create_schema ----->	create_schema
start Mnesia	
creating table ----->	???
creating table ----->	???
stop Mnesia	

For the tables to be created on all nodes, Mnesia needs to run on all nodes. For the schema to be created, Mnesia needs to run on no nodes. Ideally, we could start Mnesia and stop it remotely. The good thing is we can. Remember the RPC module from the [Distribunomicon](#)? We have the function `rpc:multicall(Nodes, Module, Function, Args)` to do it for us. Let's change the `install/1` function definition to this one:

```
install(Nodes) ->
    ok = mnesia:create_schema(Nodes),
```

```

rpc:multicall(Nodes, application, start, [mnesia]),
mnesia:create_table(mafiapp_friends,
    [{attributes, record_info(fields, mafiapp_friends)},
     {index, [#mafiapp_friends.expertise]},
     {disc_copies, Nodes}]),
mnesia:create_table(mafiapp_services,
    [{attributes, record_info(fields, mafiapp_services)},
     {index, [#mafiapp_services.to]},
     {disc_copies, Nodes},
     {type, bag}]),
rpc:multicall(Nodes, application, stop, [mnesia]).
```

Using RPC allows us to do the Mnesia action on all nodes. The scheme now looks like this:

Node A	Node B
-----	-----
create_schema ----->	create_schema
start Mnesia ----->	start Mnesia
creating table ----->	replicating table
creating table ----->	replicating table
stop Mnesia ----->	stop Mnesia

Good, very good.

The next part of the `init_per_suite/1` function we have to take care of is starting `mafiapp`. Properly speaking, there is no need to do it because our entire application depends on Mnesia: starting Mnesia is starting our application. However, there can be a noticeable delay between the time Mnesia starts and the time it finishes loading all tables from disk, especially if they're large. In such circumstances, a function such as `mafiapp's start/2` might be the perfect place to do that kind of waiting, even if we need no process at all for normal operations.

We'll make `mafiapp.erl` implement the application behaviour (`-behaviour(application).`) and add the two following callbacks in the file (remember to export them):

```
start(normal, []) ->
    mnesia:wait_for_tables([mafiapp_friends,
                            mafiapp_services], 5000),
    mafiapp_sup:start_link().
```

```
stop(_) -> ok.
```

The secret is the `mnesia:wait_for_tables(TableList, TimeOut)` function. This one will wait for at most 5 seconds (an arbitrary number, replace it with what you think fits your data) or until the tables are available.

This doesn't tell us much regarding what the supervisor should do, but that's because `mafiapp_sup` doesn't have much to do at all:

```
-module(mafiapp_sup).
-behaviour(supervisor).
-export([start_link/0]).
-export([init/1]).
```

```
start_link() ->
    supervisor:start_link(?MODULE, []).
```

```
%% This does absolutely nothing, only there to
%% allow to wait for tables.
init([]) ->
    {ok, {{one_for_one, 1, 1}, []}}.
```

The supervisor does nothing , but because the starting of OTP applications is synchronous, it's actually one of the best places to put such synchronization points.

Last, add the following `mafiapp.app` file in the `ebin/` directory to make sure the application can be started:

```
{application, mafiapp,
[{description, "Help the boss keep track of his friends"},
 {vsn, "1.0.0"}, {modules, [mafiapp, mafiapp_sup]}, {applications, [stdlib, kernel, mnesia]}]}.
```

We're now ready to write actual tests and implement our application. Or are we?

Access And Context

It might be worthwhile to have an idea of how to use Mnesia to work with tables before getting to the implementation of our app.

All modifications or even reads to a database table need to be done in something called *activity access context*. Those are different types of transactions or 'ways' to run queries. Here are the options:

transaction

A Mnesia transaction allows to run a series of database operations as a single functional block. The whole block will run on all nodes or none of them; it succeeds entirely or fails entirely. When the transaction returns, we're guaranteed that the tables were left in a consistent state, and that different transactions didn't interfere with each other, even if they tried to manipulate the same data.

This type of activity context is partially asynchronous: it will be synchronous for operations on the local node, but it will only wait for the confirmation from other nodes that they *will* commit the transaction, not that they *have* done it. The way Mnesia works, if the transaction worked locally and everyone else agreed to do it, it should work everywhere else. If it doesn't, possibly due to failures in the network or hardware, the transaction will be reverted at a later point in time; the protocol tolerates this for some efficiency reasons, but might give you confirmation that a transaction succeeded when it will be rolled back later.

sync_transaction

This activity context is pretty much the same as transaction, but it is synchronous. If the guarantees of transaction aren't enough for you

because you don't like the idea of a transaction telling you it succeeded when it may have failed due to weird errors, especially if you want to do things that have side effects (like notifying external services, spawning processes, and so on) related to the transaction's success, using `sync_transaction` is what you want. Synchronous transactions will wait for the final confirmation for all other nodes before returning, making sure everything went fine 100% of the way.

An interesting use case is that if you're doing a lot of transactions, enough to overload other nodes, switching to a synchronous mode should force things go at a slower pace with less backlog accumulation, pushing the problem of overload up a level in your application.

`async_dirty`

The `async_dirty` activity context basically bypasses all the transaction protocols and locking activities (note that it will, however, wait for active transactions to finish before proceeding). It will however keep on doing everything that includes logging, replication, etc. An `async_dirty` activity context will try to perform all actions locally, and then return, leaving other nodes' replication take place asynchronously.

`sync_dirty`

This activity context is to `async_dirty` what `sync_transaction` was to transaction. It will wait for the confirmation that things went fine on remote nodes, but will still stay out of all locking or transaction contexts. Dirty contexts are generally faster than transactions, but absolutely riskier by design. Handle with care.

`ets`

The last possible activity context is `ets`. This is basically a way to bypass everything Mnesia does and do series of raw operations on the

underlying ETS tables, if there are any. No replication will be done. The ets activity context isn't something you usually *need* to use, and thus you shouldn't want to use it. It's yet another case of "if in doubt, don't use it, and you'll know when you need it."

These are all the contexts within which common Mnesia operations can be run. These operations themselves are to be wrapped in a fun and executed by calling `mnesia:activity(Context, Fun)..` The fun can contain any Erlang function call, though be aware that it is possible for a transaction to be executed many times in case of failures or interruption by other transactions.

This means that if a transaction that reads a value from a table also sends a message before writing something back in, it is entirely possible for the message to be sent dozens of times. As such, *no side effects of the kind should be included in the transaction.*



Reads, Writes, and More

I've referred to these table-modifying functions a lot and it is now time to define them. Most of them are unsurprisingly similar to what ETS and DETS gave us.

write

By calling `mnesia:write(Record)`, where the name of the record is the name of the table, we're able to insert *Record* in the table. If the table is of type `set` or `ordered_set` and the primary key (the second field of the record, not its name, under a tuple form), the element will be replaced. For `bag` tables, the whole record will need to be similar.

If the write operation is successful, `write/1` will return `ok`. Otherwise it throws an exception that will abort the transaction. Throwing such an exception shouldn't be something frequent. It should mostly happen when Mnesia is not running, the table cannot be found, or the record is invalid.

delete

The function is called as `mnesia:delete(TableName, Key)`. The record(s) that share this key will be removed from the table. It either returns `ok` or throws an exception, with semantics similar to `mnesia:write/1`.

read

Called as `mnesia:read({TableName, Key})`, this function will return a list of records with their primary key matching `Key`. Much like `ets:lookup/2`, it will always return a list, even with tables of type `set` that can never have more than one result that matches the key. If no record matches, an empty list is returned. Much like it is done for delete and write operations, in case of a failure, an exception is thrown.

match_object

This function is similar to ETS' `match_object` function. It uses patterns such as those described in [Meeting Your Match](#) to return entire records from the database table. For example, a quick way to look for friends with a given expertise could be done with `mnesia:match_object(#mafiapp_friends{_ = '_', expertise = given})`. It will then return a list of all matching entries in the table. Once again, failures end up in exceptions being thrown.

select

This is similar to the ETS `select` function. It works using match specifications or `ets:fun2ms` as a way to do queries. If you don't remember how this works, I recommend you look back at [You Have Been Selected](#) to brush up on your matching skills. The function can be

called as `mnesia:select(TableName, MatchSpec)`, and it will return a list of all items that fit the match specification. And again, in case of failure, an exception will be thrown.

Other Operations

There are many other operations available for Mnesia tables. However, those explained before constitute a solid base for us to move forward. If you're interested in other operations, you can head to the Mnesia reference manual to find functions such as `first`, `last`, `next`, `prev` for individual iterations, `foldl` and `foldr` for folds over entire tables, or other functions to manipulate tables themselves such as `transform_table` (especially useful to add or remove fields to a record and a table) or `add_table_index`.

That makes for a lot of functions. To see how to use them realistically, we'll drive the tests forward a bit.

Implementing The First Requests

To implement the requests, we'll first write a somewhat simple test demonstrating the behavior we'll want from our application. The test will be about adding services, but will contain implicit tests for more functionality:

```
[...]
-export([init_per_suite/1, end_per_suite/1,
        init_per_testcase/2, end_per_testcase/2,
        all/0]).
-export([add_service/1]).
```

```
all() -> [add_service].
[...]
```

```
init_per_testcase(add_service, Config) ->
    Config.
```

```
end_per_testcase(_, _Config) ->
    ok.
```

This is the standard initialization stuff we need to add in most CT suites. Now for the test itself:

```
%% services can go both way: from a friend to the boss, or
%% from the boss to a friend! A boss friend is required!
add_service(_Config) ->
    {error, unknown_friend} = mafiapp:add_service("from name",
                                                    "to name",
                                                    {1946,5,23},
                                                    "a fake service"),
    ok = mafiapp:add_friend("Don Corleone", [], [boss], boss),
    ok = mafiapp:add_friend("Alan Parsons",
                            [{twitter,"@ArtScienceSound"}],
                            [{born, {1948,12,20}}],
                            musician, 'audio engineer',
                            producer, "has projects"],
                            mixing),
    ok = mafiapp:add_service("Alan Parsons", "Don Corleone",
                            {1973,3,1},
                            "Helped release a Pink Floyd album").
```

Because we're adding a service, we should add both of the friends that will be part of the exchange. The function `mafiapp:add_friend(Name, Contact, Info, Expertise)` is going to be used for that. Once the friends are added, we can actually add the service.

Note: If you've ever read other Mnesia tutorials, you'll find that some people are very eager to use records directly in the functions (say `mafiapp:add_friend(#mafiapp_friend{name=...})`). This is something that this guide tries to actively avoid as records are often better kept private. Changes in implementation might break the underlying record representation. This is not a problem in itself, but whenever you'll be changing the record definition, you'll need to recompile and, if possible, atomically update all modules that use that record so that they can keep working in a running application.

Simply wrapping things in functions gives a somewhat cleaner interface that won't require any module using your database or application to include records through .hrl files, which is frankly annoying.

You'll note that the test we just defined doesn't actually look for services. This is because what I actually plan on doing with the application is to instead search for them when looking up users. For now we can try to implement the functionality required for the test above using Mnesia transactions. The first function to be added to mafiapp.erl will be used to add a user to the database:

```
add_friend(Name, Contact, Info, Expertise) ->
    F = fun() ->
        mnesia:write(#mafiapp_friends{name=Name,
                                         contact=Contact,
                                         info=Info,
                                         expertise=Expertise})
    end,
    mnesia:activity(transaction, F).
```

We're defining a single function that writes the record `#mafiapp_friends{}`. This is a somewhat simple transaction. `add_services/4` should be a little bit more complex:

```
end,  
mnesia:activity(transaction,F).
```

You can see that in the transaction, I first do one or two reads to try to see if the friends we're trying to add are to be found in the database. If either friend is not there, the tuple {error, unknown_friend} is returned, as per the test specification. If both members of the transaction are found, we'll instead write the service to the database.

Note: validating the input is left to your discretion. Doing so requires only writing custom Erlang code like anything else you'd be programming with the language. If it is possible, doing as much validation as possible outside of the transaction context is a good idea. Code in the transaction might run many times and compete for database resources. Doing as little as possible there is always a good idea.

Based on this, we should be able to run the first test batch. To do so, I'm using the following test specification, mafiapp.spec (placed at the root of the project):

```
{alias, root, "./test/"},  
{logdir, "./logs/"},  
{suites, root, all}.
```

And the following Emakefile (also at the root):

```
[{"src/*", "test/*"},  
 [{i,"include"}, {outdir, "ebin"}]}].
```

Then, we can run the tests:

```
$ erl -make  
Recompile: src/mafiapp_sup  
Recompile: src/mafiapp  
$ ct_run -pa ebin/ -spec mafiapp.spec  
...  
Common Test: Running make in test directories...
```

```
Recompile: mafiapp_SUITE
...
Testing learn-you-some-erlang.wiptests: Starting test, 1 test cases
...
Testing learn-you-some-erlang.wiptests: TEST COMPLETE, 1 ok, 0 failed of 1 test cases
...
```

Alright, it passes. That's good. On to the next test.

Note: when running the CT suite, you might get errors saying that some directories are not found. solution is to use `ct_run -pa ebin/` or to use `erl -name ct -pa `pwd`/ebin` (or full paths). While starting the Erlang shell makes the current working directory the node's current working directory, calling `ct:run_test/1` changes the current working directory to a new one. This breaks relative paths such as `./ebin/`. Using absolute paths solves the problem.

The `add_service/1` test lets us add both friends and services. The next tests should focus on making it possible to look things up. For the sake of simplicity, we'll add the boss to all possible future test cases:

```
init_per_testcase(add_service, Config) ->
    Config;
init_per_testcase(_, Config) ->
    ok = mafiapp:add_friend("Don Corleone", [], [boss], boss),
    Config.
```

The use case we'll want to emphasize is looking up friends by their name. While we could very well search through services only, in practice we might want to look up people by name more than actions. Very rarely will the boss ask "who delivered that guitar to whom, again?" No, he'd more likely ask "Who is it who delivered the guitar to our friend Pete Cityshend?" and try to look up his history through his name to find details about the service.

As such, the next test is going to be `friend_by_name/1`:

```

-export([add_service/1, friend_by_name/1]).

all() -> [add_service, friend_by_name].
...
friend_by_name(_Config) ->
    ok = mafiapp:add_friend("Pete Cityshend",
        [{phone, "418-542-3000"}, {email, "quadrophonia@example.org"}, {other, "yell real loud"}], [{born, {1945,5,19}}], [musician, popular], music),
    {"Pete Cityshend", _Contact, _Info, music, _Services} = mafiapp:friend_by_name("Pete Cityshend"),
    undefined = mafiapp:friend_by_name(make_ref()).

```

This test verifies that we can insert a friend and look him up, but also what should be returned when we know no friend by that name. We'll have a tuple structure returning all kinds of details, including services, which we do not care about for now — we mostly want to find people, although duplicating the info would make the test stricter.

The implementation of `mafiapp:friend_by_name/1` can be done using a single Mnesia read. Our record definition for `#mafiapp_friends{}` put the friend name as the primary key of the table (first one defined in the record). By using `mnesia:read({Table, Key})`, we can get things going easily, with minimal wrapping to make it fit the test:

```

friend_by_name(Name) ->
    F = fun() ->
        case mnesia:read({mafiapp_friends, Name}) of
            [#mafiapp_friends{contact=C, info=I, expertise=E}] ->
                {Name,C,I,E,find_services(Name)};
            [] ->
                undefined
        end

```

```
end,  
mnesia:activity(transaction, F).
```

This function alone should be enough to get the tests to pass, as long as you remember to export it. We do not care about `find_services(Name)` for now, so we'll just stub it out:

```
%%% PRIVATE FUNCTIONS  
find_services(_Name) -> undefined.
```

That being done, the new test should also pass:

```
$ erl -make  
...  
$ ct_run -pa ebin/ -spec mafiapp.spec  
...  
Testing learn-you-some-erlang.wiptests: TEST COMPLETE, 2 ok, 0 failed of 2 test cases  
...
```

It would be nice to put a bit more details into the services area of the request. Here's the test to do it:

```
-export([add_service/1, friend_by_name/1, friend_with_services/1]).
```

```
all() -> [add_service, friend_by_name, friend_with_services].  
...  
friend_with_services(_Config) ->  
    ok = mafiapp:add_friend("Someone", [{other, "at the fruit stand"}],  
                            [weird, mysterious], shadiness),  
    ok = mafiapp:add_service("Don Corleone", "Someone",  
                            {1949,2,14}, "Increased business"),  
    ok = mafiapp:add_service("Someone", "Don Corleone",  
                            {1949,12,25}, "Gave a Christmas gift"),  
    %% We don't care about the order. The test was made to fit  
    %% whatever the functions returned.  
    {"Someone",  
     _Contact, _Info, shadiness,  
     [{to, "Don Corleone", {1949,12,25}, "Gave a Christmas gift"},  
      {from, "Don Corleone", {1949,2,14}, "Increased business"}]} =  
    mafiapp:friend_by_name("Someone").
```

In this test, Don Corleone helped a shady person with a fruit stand to grow his business. Said shady person at the fruit stand later gave a Christmas gift to the boss, who never forgot about it.

You can see that we still use `friend_by_name/1` to search entries. Although the test is overly generic and not too complete, we can probably figure out what we want to do; fortunately, the total absence of maintainability requirements kind of makes it okay to do something this incomplete.

The `find_service/1` implementation will need to be a bit more complex than the previous one. While `friend_by_name/1` could work just by querying the primary key, the friends names in services is only the primary key when searching in the `from` field. We still need to deal with the `to` field. There are many ways to handle this one, like using `match_object` many times or reading the entire table and filtering things manually. I chose to use Match Specifications and the `ets:fun2ms/1` parse transform:

```
-include_lib("stdlib/include/ms_transform.hrl").  
...  
find_services(Name) ->  
    Match = ets:fun2ms(  
        fun(#mafiapp_services{from=From, to=To, date=D, description=Desc})  
            when From =:= Name ->  
                {to, To, D, Desc};  
            (#mafiapp_services{from=From, to=To, date=D, description=Desc})  
            when To =:= Name ->  
                {from, From, D, Desc}  
        end  
>),  
    mnesia:select(mafiapp_services, Match).
```

This match specification has two clauses: whenever `From` matches `Name` we return a `{to, ToName, Date, Description}` tuple. Whenever `Name` matches `To` instead, the function returns a tuple of the form `{from,`

`FromName, Date, Description}`, allowing us to have a single operation that includes both services given and received.

You'll note that `find_services/1` does not run in any transaction. That's because the function is only called within `friend_by_name/1`, which runs in a transaction already. Mnesia can in fact run nested transactions, but I chose to avoid it because it was useless to do so in this case.

Running the tests again should reveal that all three of them do, in fact, work.

The last use case we had planned for was the idea of searching for friends through their expertise. The following test case, for example, illustrates how we might find our friend the red panda when we need climbing experts for some task:

```
-export([add_service/1, friend_by_name/1, friend_with_services/1,
        friend_by_expertise/1]).  
  
all() -> [add_service, friend_by_name, friend_with_services,  
          friend_by_expertise].  
  
...  
friend_by_expertise(_Config) ->  
    ok = mafiapp:add_friend("A Red Panda",  
                           [{location, "in a zoo"}],  
                           [animal,cute],  
                           climbing),  
    [{"A Red Panda",  
     _Contact, _Info, climbing,  
     _Services}] = mafiapp:friend_by_expertise(climbing),  
    [] = mafiapp:friend_by_expertise(make_ref()).
```

To implement that one, we'll need to read something else than the primary key. We could use match specifications for that one, but we've already done that. Plus, we only need to match on one field. The `mnesia:match_object/1` function is well adapted to this:

```

friend_by_expertise(Expertise) ->
    Pattern = #mafiapp_friends{_ = '_',
                                expertise = Expertise},
    F = fun() ->
        Res = mnesia:match_object(Pattern),
        [{Name,C,I,Expertise,find_services(Name)} ||
         #mafiapp_friends{name=Name,
                           contact=C,
                           info=I} <- Res]
    end,
    mnesia:activity(transaction, F).

```

In this one, we first declare the pattern. We need to use `_ = '_'` to declare all undefined values as a match-all specification (`'_'`). Otherwise, the `match_object/1` function will look only for entries where everything but the expertise is the atom undefined.

Once the result is obtained, we format the record into a tuple, in order to respect the test. Again, compiling and running the tests will reveal that this implementation works. Hooray, we implemented the whole specification!

Accounts And New Needs

No software project is ever really done. Users using the system bring new needs to light or break it in unexpected ways. The Boss, even before using our brand new software product, decided that he wants a feature letting us quickly go through all of our friends and see who we owe things to, and who actually owes us things.

Here's the test for that one:

```

...
init_per_testcase(accounts, Config) ->
    ok = mafiapp:add_friend("Consigliere", [], [you], consigliere),
    Config;
...
accounts(_Config) ->

```

```

ok = mafiapp:add_friend("Gill Bates", [{email, "ceo@macrohard.com"}],
                         [clever,rich], computers),
ok = mafiapp:add_service("Consigliere", "Gill Bates",
                         {1985,11,20}, "Bought 15 copies of software"),
ok = mafiapp:add_service("Gill Bates", "Consigliere",
                         {1986,8,17}, "Made computer faster"),
ok = mafiapp:add_friend("Pierre Gauthier", [{other, "city arena"}],
                         [{job, "sports team GM"}], sports),
ok = mafiapp:add_service("Pierre Gauthier", "Consigliere", {2009,6,30},
                         "Took on a huge, bad contract"),
ok = mafiapp:add_friend("Wayne Gretzky", [{other, "Canada"}],
                         [{born, {1961,1,26}}], "hockey legend"),
                         hockey),
ok = mafiapp:add_service("Consigliere", "Wayne Gretzky", {1964,1,26},
                         "Gave first pair of ice skates"),
%% Wayne Gretzky owes us something so the debt is negative
%% Gill Bates are equal
%% Gauthier is owed a service.
[{-1,"Wayne Gretzky"},  

{0,"Gill Bates"},  

{1,"Pierre Gauthier"}] = mafiapp:debts("Consigliere"),  

[{1, "Consigliere"}] = mafiapp:debts("Wayne Gretzky").

```

We're adding three test friends in the persons of Gill Bates, Pierre Gauthier, and hockey hall of famer Wayne Gretzky. There is an exchange of services going between each of them and you, the consigliere (we didn't pick the boss for that test because he's being used by other tests and it would mess with the results!)

The mafiapp:debts(Name) function looks for a name, and counts all the services where the name is involved. When someone owes us something, the value is negative. When we're even, it's 0, and when we owe something to someone, the value is one. We can thus say that the debt/1 function returns the number of services owed to different people.

The implementation of that function is going to be a bit more complex:

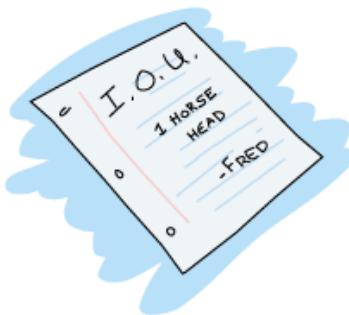
```

-export([install/1, add_friend/4, add_service/4, friend_by_name/1,
        friend_by_expertise/1, debts/1]).

...
debts(Name) ->
    Match = ets:fun2ms(
        fun(#mafiapp_services{from=From, to=To}) when From =:= Name ->
            {To,-1};
        (#mafiapp_services{from=From, to=To}) when To =:= Name ->
            {From,1}
    end),
    F = fun() -> mnesia:select(mafiapp_services, Match) end,
    Dict = lists:foldl(fun({Person,N}, Dict) ->
        dict:update(Person, fun(X) -> X + N end, N, Dict)
    end,
    dict:new(),
    mnesia:activity(transaction, F)),
    lists:sort([{V,K} || {K,V} <- dict:to_list(Dict)]).

```

Whenever Mnesia queries get to be complex, match specifications are usually going to be part of your solution. They let you run basic Erlang functions and they thus prove invaluable when it comes to specific result generation. In the function above, the match specification is used to find that whenever the service given comes from *Name*, its value is -1 (we gave a service, they owe us one). When *Name* matches *To*, the value returned will be 1 (we received a service, we owe one). In both cases, the value is coupled to a tuple containing the name.



Including the name is necessary for the second step of the computation, where we'll try to count all the services given for each

person and give a unique cumulative value. Again, there are many ways to do it. I picked one that required me to stay as little time as possible within a transaction to allow as much of my code to be separated from the database. This is useless for mafiapp, but in high performance cases, this can reduce the contention for resources in major ways.

Anyway, the solution I picked is to take all the values, put them in a dictionary, and use dictionaries' `dict:update(Key, Operation)` function to increment or decrement the value based on whether a move is for us or from us. By putting this into a fold over the results given by Mnesia, we get a list of all the values required.

The final step is to flip the values around (from `{Key,Debt}` to `{Debt, Key}`) and sort based on this. This will give the results desired.

Meet The Boss

Our software product should at least be tried once in a production. We'll do this by setting up the node the boss will use, and then yours.

```
$ erl -name corleone -pa ebin/
```

```
$ erl -name genco -pa ebin/
```

Once both nodes are started, you can connect them and install the app:

```
(corleone@ferdmbp.local)1> net_kernel:connect_node('genco@ferdmbp.local').  
true  
(corleone@ferdmbp.local)2> mafiapp:install([node()|nodes()]).  
{[ok,ok],[]}  
(corleone@ferdmbp.local)3>  
=INFO REPORT==== 8-Apr-2012::20:02:26 ====  
    application: mnesia  
    exited: stopped  
    type: temporary
```

You can then start running Mnesia and Mafiapp on both nodes by calling application:start(mnesia), application:start(mafiapp). Once it's done, you can try and see if everything is running fine by calling mnesia:system_info(), which will display status information about your whole setup:

```
(genco@ferdmbp.local)2> mnesia:system_info().
==> System info in version "4.7", debug level = none <==
opt_disc. Directory "/Users/ferd/.../Mnesia.genco@ferdmbp.local" is used.
use fallback at restart = false
running db nodes  = ['corleone@ferdmbp.local','genco@ferdmbp.local']
stopped db nodes  = []
master node tables = []
remote      = []
ram_copies   = []
disc_copies  = [mafiapp_friends,mafiapp_services,schema]
disc_only_copies = []
[{'corleone@...',disc_copies},{'genco@...',disc_copies}] = [schema,
                                         mafiapp_friends,
                                         mafiapp_services]
5 transactions committed, 0 aborted, 0 restarted, 2 logged to disc
0 held locks, 0 in queue; 0 local transactions, 0 remote
0 transactions waits for other nodes: []
yes
```

You can see that both nodes are in the running DB nodes, that both tables and the schema are written to disk and in RAM (disc_copies). We can start writing and reading data from the database. Of course, getting the Don part inside the DB is a good starting step:

```
(corleone@ferdmbp.local)4> ok = mafiapp:add_friend("Don Corleone", [], [boss], boss).
ok
(corleone@ferdmbp.local)5> mafiapp:add_friend(
(corleone@ferdmbp.local)5>   "Albert Einstein",
(corleone@ferdmbp.local)5>   [{city, "Princeton, New Jersey, USA"}],
(corleone@ferdmbp.local)5>   [physicist, savant,
(corleone@ferdmbp.local)5>     [{awards, [{1921, "Nobel Prize"}]}]],
(corleone@ferdmbp.local)5>   physicist).
ok
```

Alright, so friends were added from the corleone node. Let's try adding a service from the genco node:

```
(genco@ferdmbp.local)3> mafiapp:add_service("Don Corleone",
(genco@ferdmbp.local)3>                      "Albert Einstein",
(genco@ferdmbp.local)3>                      {1905, '?', '?'},
(genco@ferdmbp.local)3>                      "Added the square to E = MC").
ok
(genco@ferdmbp.local)4> mafiapp:debts("Albert Einstein").
[{"Don Corleone"}]
```

And all these changes can also be reflected back to the corleone node:

```
(corleone@ferdmbp.local)6> mafiapp:friend_by_expertise(physicist).
[{"Albert Einstein",
 [{"city,"Princeton, New Jersey, USA"}],
 [physicist,savant,[{awards,[{1921,"Nobel Prize"}]}]],
 physicist,
 [{"from,"Don Corleone",
   {1905,'?','?'},
   "Added the square to E = MC"}]]
```

Great! Now if you shut down one of the nodes and start it again, things should still be fine:

```
(corleone@ferdmbp.local)7> init:stop().
ok

$ erl -name corleone -pa ebin
...
(corleone@ferdmbp.local)1> net_kernel:connect_node('genco@ferdmbp.local').
true
(corleone@ferdmbp.local)2> application:start(mnesia), application:start(mafiapp).
ok
(corleone@ferdmbp.local)3> mafiapp:friend_by_expertise(physicist).
[{"Albert Einstein",
 ...
   "Added the square to E = MC"}]]
```

Isn't it nice? We're now knowledgeable about Mnesia!

Note: if you end up working on a system where tables begin being messy or are just curious about looking at entire tables, call the function `observer:start()`. It will start a graphical interface with table viewer tab letting you interact with tables visually, rather than through code. On older Erlang releases where the `observer` application did not yet exist, calling `tv:start()` will start its predecessor.

Deleting Stuff, Demonstrated

Wait. Did we just entirely skip over *deleting* records from a database? Oh no! Let's add a table for that.

We'll do it by creating a little feature for you and the boss that lets you store your own personal enemies, for personal reasons:

```
-record(mafiapp_enemies, {name,  
    info=[]}).
```

Because this will be personal enemies, we'll need to install the table by using slightly different table settings, using `local_content` as an option when installing the table. This will let the table be private to each node, so that nobody can read anybody else's personal enemies accidentally (although RPC would make it trivial to circumvent).

Here's the new install function, preceded by mafiapp's `start/2` function, changed for the new table:

```
start(normal, []) ->  
    mnesia:wait_for_tables([mafiapp_friends,  
        mafiapp_services,  
        mafiapp_enemies], 5000),  
    mafiapp_sup:start_link().  
  
...  
install(Nodes) ->  
    ok = mnesia:create_schema(Nodes),  
    application:start(mnesia),  
    mnesia:create_table(mafiapp_friends,  
        [{attributes, record_info(fields, mafiapp_friends)}],
```

```

{index, [#mafiapp_friends.expertise]},
{disc_copies, Nodes]}),
mnesia:create_table(mafiapp_services,
[{attributes, record_info(fields, mafiapp_services)},
{index, [#mafiapp_services.to]},
{disc_copies, Nodes},
{type, bag}]),
mnesia:create_table(mafiapp_enemies,
[{attributes, record_info(fields, mafiapp_enemies)},
{disc_copies, Nodes},
{local_content, true}]),
application:stop(mnesia).

```

The start/2 function now sends mafiapp_enemies through the supervisor to keep things alive there. The install/1 function will be useful for tests and fresh installs, but if you're doing things in production, you can straight up call mnesia:create_table/2 in production to add tables. Depending on the load on your system and how many nodes you have, you might want to have a few practice runs in staging first, though.

Anyway, this being done, we can write a simple test to work with our DB and see how it goes, still in mafiapp_SUITE:

```

...
-export([add_service/1, friend_by_name/1, friend_by_expertise/1,
friend_with_services/1, accounts/1, enemies/1]).

all() -> [add_service, friend_by_name, friend_by_expertise,
friend_with_services, accounts, enemies].

...
enemies(_Config) ->
undefined = mafiapp:find_enemy("Edward"),
ok = mafiapp:add_enemy("Edward", [{bio, "Vampire"}, {comment, "He sucks (blood)"}]),
>{"Edward", [{bio, "Vampire"}, {comment, "He sucks (blood)"}]} =
mafiapp:find_enemy("Edward"),
ok = mafiapp:enemy_killed("Edward"),
undefined = mafiapp:find_enemy("Edward").

```

This is going to be similar to previous runs for `add_enemy/2` and `find_enemy/1`. All we'll need to do is a basic insertion for the former and a `mnesia:read/1` based on the primary key for the latter:

```
add_enemy(Name, Info) ->
    F = fun() -> mnesia:write(#mafiapp_enemies{name=Name, info=Info}) end,
    mnesia:activity(transaction, F).
```

```
find_enemy(Name) ->
    F = fun() -> mnesia:read({mafiapp_enemies, Name}) end,
    case mnesia:activity(transaction, F) of
        [] -> undefined;
        [#mafiapp_enemies{name=N, info=I}] -> {N,I}
    end.
```

The `enemy_killed/1` function is the one that's a bit different:

```
enemy_killed(Name) ->
    F = fun() -> mnesia:delete({mafiapp_enemies, Name}) end,
    mnesia:activity(transaction, F).
```

And that's pretty much it for basic deletions. You can export the functions, run the test suite and all the tests should still pass.

When trying it on two nodes (after deleting the previous schemas, or possibly just calling the `create_table` function), we should be able to see that data between tables isn't shared:

```
$ erl -name corleone -pa ebin
```

```
$ erl -name genco -pa ebin
```

With the nodes started, I reinstall the DB:

```
(corleone@ferdmbp.local)1> net_kernel:connect_node('genco@ferdmbp.local').
true
(corleone@ferdmbp.local)2> mafiapp:install([node()|nodes()]).
```

```
=INFO REPORT==== 8-Apr-2012::21:21:47 ===
```

...
{[ok,ok],[]}

Start the apps and get going:

```
(genco@ferdmbp.local)1> application:start(mnesia), application:start(mafiapp).  
ok  
  
(corleone@ferdmbp.local)3> application:start(mnesia), application:start(mafiapp).  
ok  
(corleone@ferdmbp.local)4> mafiapp:add_enemy("Some Guy", "Disrespected his family").  
ok  
(corleone@ferdmbp.local)5> mafiapp:find_enemy("Some Guy").  
{"Some Guy","Disrespected his family"}  
  
(genco@ferdmbp.local)2> mafiapp:find_enemy("Some Guy").  
undefined
```

And you can see, no data shared. Deleting the entry is also as simple:

```
(corleone@ferdmbp.local)6> mafiapp:enemy_killed("Some Guy").  
ok  
(corleone@ferdmbp.local)7> mafiapp:find_enemy("Some Guy").  
undefined
```

Finally!

Query List Comprehensions

If you've silently followed this chapter (or worse, skipped right to this part!) thinking to yourself "Damn, I don't like the way Mnesia looks", you might like this section. If you liked how Mnesia looked, you might also like this section. Then if you like list comprehensions, you'll definitely like this section too.

Query List Comprehensions are basically a compiler trick using parse transforms that let you use list comprehensions for any data structure that can be searched and iterated through. They're implemented for

Mnesia, DETS, and ETS, but can also be implemented for things like gb_trees.

Once you add `-include_lib("stdlib/include/qlc.hrl")`. to your module, you can start using list comprehensions with something called a *query handle* as a generator. The query handle is what allows any iterable data structure to work with QLC. In the case of Mnesia, what you can do is use `mnesia:table(TableName)` as a list comprehension generator, and from that point on, you can use list comprehensions to query any database table by wrapping them in a call to `qlc:q(...)`.

This will in turn return a modified query handle, with more details than the one returned by the table. This newest one can subsequently be modified some more by using functions like `qlc:sort/1-2`, and can be evaluated by using `qlc:eval/1` or `qlc:fold/1`.

Let's get straight to practice with it. We'll rewrite a few of the mafiapp functions. You can make a copy of mafiapp-1.0.0 and call it mafiapp-1.0.1 (don't forget to bump the version in the .app file).

The first function to rework will be `friend_by_expertise`. That one is currently implemented using `mnesia:match_object/1`. Here's a version using QLC:

```
friend_by_expertise(Expertise) ->
    F = fun() ->
        qlc:eval(qlc:q(
            [{Name,C,I,E,find_services(Name)} ||
             #mafiapp_friends{name = Name,
                               contact = C,
                               info = I,
                               expertise = E} <- mnesia:table(mafiapp_friends),
             E =:= Expertise]))
    end,
    mnesia:activity(transaction, F).
```

You can see that except for the part where we call `qlc:eval/1` and `qlc:q/1`, this is a normal list comprehension. You have the final expression in

{Name,C,I,E,find_services(Name)}, the generator in #mafiapp{...} <- mnesia:table(...), and finally, a condition with E ==:= Expertise. Searching through database tables is now a bit more natural, Erlang-wise.

That's pretty much all there is to query list comprehensions. Really. However, I think we should try a slightly bit more complex example. Let's take a look at the debts/1 function. It was implemented using a match specification and then a fold over to a dictionary. Here's how that could look using QLC:

```
debts(Name) ->
    F = fun() ->
        QH = qlc:q(
            [if Name ==:= To -> {From,1};
             Name ==:= From -> {To,-1}
            end || #mafiapp_services{from=From, to=To} <-
                  mnesia:table(mafiapp_services),
                  Name ==:= To orelse Name ==:= From]),
            qlc:fold(fun({Person,N}, Dict) ->
                dict:update(Person, fun(X) -> X + N end, N, Dict)
                end,
                dict:new(),
                QH)
            end,
        lists:sort([{V,K} || {K,V} <- dict:to_list(mnesia:activity(transaction, F))]).
```

The match specification is no longer necessary. The list comprehension (saved to the *QH* query handle) does that part. The fold has been moved into the transaction and is used as a way to evaluate the query handle. The resulting dictionary is the same as the one that was formerly returned by lists:foldl/3. The last part, sorting, is handled outside of the transaction by taking whatever dictionary mnesia:activity/1 returned and converting it to a list.

And there you go. If you write these functions in your mafiapp-1.0.1 application and run the test suite, all 6 tests should still pass.



Remember Mnesia

That's it for Mnesia. It's quite a complex database and we've only seen a moderate portion of everything it can do. Pushing further ahead will require you to read the Erlang manuals and dive into the code. The programmers that have true production experience with Mnesia in large, scalable systems that have been running for years are rather rare. You can find a few of them on mailing lists, sometimes answering a few questions, but they're generally busy people.

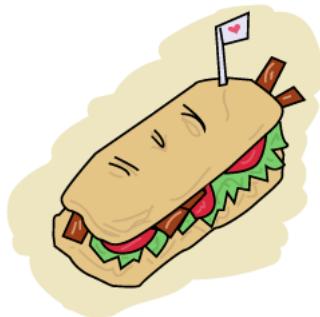
Otherwise, Mnesia is always a very nice tool for smaller applications where you find picking a storage layer to be very annoying, or even larger ones where you will have a known number of nodes, as mentioned earlier. Being able to store and replicate Erlang terms directly is a very neat thing — something other languages tried to write for years using Object-Relational Mappers.

Interestingly enough, someone putting his mind to it could likely write QLC selectors for SQL databases or any other kind of storage that allows iteration.

Mnesia and its tool chain have all the potential to be very useful in some of your future applications. For now though, we'll move to additional tools to help you develop Erlang systems with Dialyzer.

Type Specifications and Erlang

PLTs Are The Best Sandwiches



Back in [Types \(or lack thereof\)](#), I introduced Dialyzer, a tool to find type errors in Erlang. This chapter puts its full focus on Dialyzer and how to actually find some type errors with Erlang, and how to use the tool to find other types of discrepancies. We'll start by seeing why Dialyzer was created, then what the guiding principles behind it are and its capabilities to find type-related errors, and finally, a few examples of it in use.

Dialyzer is a very effective tool when it comes to analyzing Erlang code. It's used to find all kinds of discrepancies, such as code that will never be executed, but its main use is usually centered around finding type errors in your Erlang code base.

Before seeing much of it, we'll create Dialyzer's *Persistent Lookup Table (PLT)*, which is a compilation of all the details Dialyzer can identify about the applications and modules that are part of your standard Erlang distribution, and code outside of OTP too. It takes quite a while to compile everything, especially if you're running it on a platform that has no native compilation through HiPE (namely Windows), or on older versions of Erlang. Fortunately things tend to get faster with time, and the newest versions of Erlang in newer releases (R15B02 onward) are getting parallel Dialyzer to make things even faster. Enter the following command into a terminal, and let it run as long as it needs:

```
$ dialyzer --build_plt --apps erts kernel stdlib crypto mnesia sasl common_test eunit
Compiling some key modules to native code... done in 1m19.99s
Creating PLT /Users/ferd/.dialyzer_plt ...
eunit_test.erl:302: Call to missing or unexported function eunit_test:nonexisting_function/0
Unknown functions:
compile:file/2
compile:forms/2
...
xref:stop/1
Unknown types:
compile:option/0
done in 6m39.15s
done (warnings were emitted)
```

This command builds the PLT by specifying which OTP applications we want to include in it. You can ignore the warnings if you want, as Dialyzer can deal with unknown functions when looking for type errors. We'll see why when we get to discuss how its type inference algorithm works, later in the chapter. Some Windows users will see an error message saying "The HOME environment variable needs to be set so that Dialyzer knows where to find the default PLT". This is because Windows doesn't always come with the HOME environment variable set, and Dialyzer doesn't know where to dump the PLT. Set the variable to wherever you want Dialyzer to place its files.

If you want, you can add applications like `ssl` or `reltool` by adding them to the sequence that follows `--apps`, or if your PLT is already built, by calling:

```
$ dialyzer --add_to_plt --apps ssl reltool
```

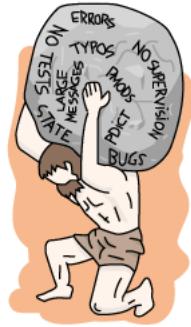
If you want to add your own applications or modules to the PLT, you can do so by using `-r` Directories, which will look for all `.erl` or `.beam` files (as long as they're compiled with `debug_info`) to add them to the PLT.

Moreover, Dialyzer lets you have many PLTs by specifying them with the `--plt Name` option in any of the commands you do, and pick a specific PLT. Alternatively, if you built many *disjoint PLTs* where none of the included modules are shared between PLTs, you can 'merge' them by using `--plts Name1 Name2 ... NameN`. This is especially useful when you want to have different PLTs in your system for different projects or different Erlang versions. The downside of this is that information obtained from merged PLTs is not as precise as if all information was contained in a single one to begin with.

While the PLT is still building, we should get acquainted with Dialyzer's mechanism to find type errors.

Success Typing

As it is the case with most other dynamic programming languages, Erlang programs are always at risk of suffering from type errors. A programmer passes in some arguments to a function he shouldn't have, and maybe he forgot to test things properly. The program gets deployed, and everything seems to be going okay. Then at four in the morning, your company's operations guy's cell phone starts ringing because your piece of software is repeatedly crashing, enough that the supervisors can't cope with the sheer weight of your mistakes.



The next morning, you get at the office, and you find your computer has been reformatted, your car gets keyed, and your commit rights have been revoked, all by the operations guy who has had enough of you accidentally controlling his work schedule.

That entire debacle could have been prevented by a compiler that has a static type analyzer to verify programs before they run. While Erlang doesn't crave a type system as much as other dynamic languages, thanks to its reactive approach to run-time errors, it is definitely nice to benefit from the additional safety usually provided by early type-related error discovery.

Usually, languages with static type systems get to be designed that way. The semantics of the language is heavily influenced by their type systems in what they allow and don't allow. For example, a function such as:

```
foo(X) when is_integer(X) -> X + 1;  
foo(X) -> list_to_atom(X).
```

Most type systems are unable to properly represent the types of the function above. They can see that it can take an integer or a list and return an integer or an atom, but they won't track the dependency between the input type of the function and its output type (conditional types and intersection types are able to, but they can be verbose). This means that writing such functions, which is entirely normal in Erlang, can result in some uncertainty for the type analyzer when these functions get used later on in the code.

Generally speaking, analyzers will want to actually *prove* that there will be no type errors at run time, as in mathematically prove. This means that in a few circumstances, the type checker will disallow certain practically valid operations for the sake of removing uncertainty that could lead to crashes.

Implementing such a type system would likely mean forcing Erlang to change its semantics. The problem is that by the time Dialyzer came around, Erlang was already well in use for very large projects. For any tool like Dialyzer to be accepted, it needed to respect Erlang's philosophies. If Erlang allows pure nonsense in its types that can only be solved at run time, so be it. The type checker doesn't have a right to complain. No programmer likes a tool that tells him his program cannot run when it has been doing so in production for a few months already!

The other option is then to have a type system that will not *prove* the absence of errors, but will do a best effort at detecting whatever it can. You can make such detection really good, but it will never be perfect. It's a tradeoff to be made.

Dialyzer's type system thus made the decision not to prove that a program is error-free when it comes to types, but only to find as many errors as possible without ever contradicting what happens in the real world:

Our main goal is to make uncover the implicit type information in Erlang code and make it explicitly available in programs. Because of the sizes of typical Erlang applications, the type inference should be completely automatic and faithfully respect the operational semantics of the language. Moreover, it should impose no code rewrites of any kind. The reason for this is simple. Rewriting, often safety critical, applications consisting of hundreds of thousand lines of code just to satisfy a type inferencer is not an option which will enjoy much success. However, large software applications have to be maintained, and often not by their original authors. By automatically revealing the type information that is already present, we provide automatic documentation that can evolve together with the program and will not rot. We also think that it is important to achieve a balance between precision and readability. Last but not least, the inferred typings should never be wrong.

As the Success Typings paper behind Dialyzer explains it, a type checker for a language like Erlang should work without type declarations being there (although it accepts hints), should be simple and readable, should adapt to the language (and not the other way around), and only complain on type errors that would guarantee a crash.

Dialyzer thus begins each analysis optimistically assuming that all functions are good. It will see them as always succeeding, accepting anything, and possibly returning anything. No matter how an unknown function is used, it's a good way to use it. This is why warnings about unknown functions are not a big deal when generating PLTs. It's all good anyway; Dialyzer is a natural optimist when it comes to type inference.

As the analysis goes, Dialyzer gets to know your functions better and better. As it does so, it can analyze the code and see some interesting things. Say one of your functions has a + operator between both of its arguments and that it returns the value of the addition. Dialyzer no longer assumes that the function takes anything and returns anything, but will now expect the arguments to be numbers (either integers or floating point values), and the returned values will similarly be numbers. This function will have a basic type associated to it saying that it accepts two numbers and returns a number.

Now let's say one of your functions calls the one described above with an atom and a number. Dialyzer will think about the code and say "wait a minute, you can't use an atom and a number with the + operator!" It will then freak out because where the function could return a number before, it can not return anything given how you use it.

In more general circumstances, though, you might see Dialyzer keep silent on many things that you know will sometimes cause an error. Take for example a snippet of code looking a bit like this:

```
main() ->
    X = case fetch() of
        1 -> some_atom;
        2 -> 3.14
    end,
    convert(X).

convert(X) when is_atom(X) -> {atom, X}.
```

This bit of code assumes the existence of a `fetch/0` function that returns either 1 or 2. Based on this, we either return an atom or a floating point number.

From our point of view, it appears that at some point in time, the call to `convert/1` will fail. We'd likely expect a type error there whenever `fetch()` returns 2, which sends a floating point value to `convert/1`. Dialyzer doesn't think so. Remember, Dialyzer is optimistic. It has figurative faith in your code, and because there is the *possibility* that the function call to `convert/1` succeeds at some point, Dialyzer will keep silent. No type error is reported in this case.

Type Inference and Discrepancies

For a practical example of the principles above, let's try Dialyzer on a few modules. The modules are `discrep1.erl`, `discrep2.erl`, and `discrep3.erl`. They're evolutions of each other. Here's the first one:

```
-module(discrep1).
-export([run/0]).

run() -> some_op(5, you).

some_op(A, B) -> A + B.
```

The error in that one is kind of obvious. You can't add 5 to the `you` atom. We can try Dialyzer on that piece of code, assuming the PLT has been created:

```
$ dialyzer discrep1.erl
Checking whether the PLT /home/ferd/.dialyzer_plt is up-to-date... yes
Proceeding with analysis...
discrep1.erl:4: Function run/0 has no local return
discrep1.erl:4: The call discrep1:some_op(5,'you') will never return since it differs in the 2nd argument from the success
discrep1.erl:6: Function some_op/2 has no local return
discrep1.erl:6: The call erlang:'+'(A::5,B::'you') will never return since it differs in the 2nd argument from the success typi
done in 0m0.62s
done (warnings were emitted)
```

Oh bloody fun, Dialyzer found stuff. What the hell does it mean? The first one is an error you will see happening a *lot* of times using Dialyzer. 'Function Name/Arity has no local return' is

the standard Dialyzer warning emitted whenever a function provably doesn't return anything (other than perhaps raising an exception) because one of the functions it calls happens to trip Dialyzer's type error detector or raises an exception itself. When such a thing happens, the set of possible types of values the function could return is empty; it doesn't actually return. This error propagates to the function that called it, giving us the 'no local return' error.

The second error is somewhat more understandable. It says that calling `some_op(5, 'you')` breaks what Dialyzer detected would be the types required to make the function work, which is two numbers (`number()` and `number()`). Granted the notation is a bit foreign for now, but we'll see it in more details soon enough.

The third error is yet again a no local return. The first one was because `some_op/2` would fail, this one is because the `+` call that will fail. This is what the fourth and last error is about. The plus operator (actually the function `erlang:'+'/2` can't add the number 5 to the atom `you`.

What about `discrep2.erl`? Here's what it looks like:

```
-module(discrep2).
-export([run/0]).

run() ->
    Tup = money(5, you),
    some_op(count(Tup), account(Tup)).

money(Num, Name) -> {give, Num, Name}.
count({give, Num, _}) -> Num.
account({give, _, X}) -> X.

some_op(A, B) -> A + B.
```

If you run Dialyzer over that file again, you'll get similar errors to before:

```
$ dialyzer discrep2.erl
Checking whether the PLT /home/ferd/.dialyzer_plt is up-to-date... yes
Proceeding with analysis...
discrep2.erl:4: Function run/0 has no local return
discrep2.erl:6: The call discrep2:some_op(5,'you') will never return since it differs in the 2nd argument from the success type
discrep2.erl:12: Function some_op/2 has no local return
discrep2.erl:12: The call erlang:'+'(A::5,B:'you') will never return since it differs in the 2nd argument from the success type
done in 0m0.69s
done (warnings were emitted)
```

During its analysis, Dialyzer can see the types right through the `count/1` and `account/1` functions. It infers the types of each of the elements of the tuple and then figures out the values they pass. It can then find the errors again, without a problem.

Let's push it a bit further, with `discrep3.erl`:

```

-module(discrep3).
-export([run/0]).

run() ->
    Tup = money(5, you),
    some_op(item(count, Tup), item(account, Tup)).

money(Num, Name) -> {give, Num, Name}.

item(count, {give, X, _}) -> X;
item(account, {give, _, X}) -> X.

some_op(A, B) -> A + B.

```

This version introduces a new level of indirection. Instead of having a function clearly defined for the count and the account values, this one works with atoms and switches to different function clauses. If we run Dialyzer on it, we get this:

```

$ dialyzer discrep3.erl
Checking whether the PLT /home/ferd/.dialyzer_plt is up-to-date... yes
Proceeding with analysis... done in 0m0.70s
done (passed successfully)

```



Uh oh. Somehow the new change to the file made things complex enough that Dialyzer got lost in our type definitions. The error is still there though. We'll get back to understanding why Dialyzer doesn't find the errors in this file and how to fix it in a while, but for now, there are still a few more ways to run Dialyzer that we need to explore.

If we wanted to run Dialyzer over, say, our Process Quest release, we could do it as follows:

```

$ cd processquest/apps
$ ls
processquest-1.0.0 processquest-1.1.0 regis-1.0.0 regis-1.1.0 sockserv-1.0.0 sockserv-1.0.1

```

So we've got a bunch of libraries. Dialyzer wouldn't like it if we had many modules with the same names, so we'll need to specify directories manually:

```

$ dialyzer -r processquest-1.1.0/src regis-1.1.0/src sockserv-1.0.1/src
Checking whether the PLT /home/ferd/.dialyzer_plt is up-to-date... yes
Proceeding with analysis...
dialyzer: Analysis failed with error:
No .beam files to analyze (no --src specified?)

```

Oh right. By default, dialyzer will look for .beam files. We need to add the --src flag to tell Dialyzer to use .erl files for its analysis:

```
$ dialyzer -r processquest-1.1.0/src regis-1.1.0/src sockserv-1.0.1/src --src  
Checking whether the PLT /home/ferd/.dialyzer_plt is up-to-date... yes  
Proceeding with analysis... done in 0m2.32s  
done (passed successfully)
```

You'll note that I chose to add the `src` directory to all requests. You could have done the same search without it, but then Dialyzer would have complained about a bunch of errors related to EUnit tests, based on how some of the assertion macros work with regard to the code analysis — we do not really care about these. Plus, if you sometimes test for failures and make your software crash on purpose inside of tests, Dialyzer will pick on that and you might not want it to.

Typing About Types of Types

As seen with `discrep3.erl`, Dialyzer will sometimes not be able to infer all the types in the way we intended it. That's because Dialyzer cannot read our minds. To help out Dialyzer in its task (and mostly help ourselves), it is possible to declare types and annotate functions in order to both document them and help formalize the implicit expectations about types we put in our code.

Erlang types can be things simple as say, the number 42, noted 42 as a type (nothing different from usual), or specific atoms such as `cat`, or maybe molecule. Those are called *singleton types* as they refer to a value itself. The following singleton types exist:

- 'some atom' Any atom can be its own singleton type.
- 42 A given integer.
- [] An empty list.
- { } An empty tuple.
- <>> An empty binary.

You can see that it could be annoying to program Erlang using only these types. There is no way to express things such as ages, or much less "all the integers" for our programs by using singleton types. And then, even if we had a way to specify many types at once, it would be awfully annoying to express things such as 'any integer' without writing them all by hand, which isn't exactly possible anyway.

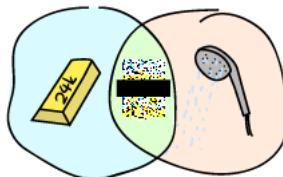
Because of this, Erlang has *union types*, which allow you to describe a type that has two atoms in it, and *built-in types*, which are pre-defined types, not necessarily possible to build by hand, and they're generally useful. Union types and built-in types generally share a similar syntax, and they're noted with the form `TypeName()`. For example, the type for all possible integers would be noted `integer()`. The reason why parentheses are used is that they let us differentiate between, say the type `atom()` for all atoms, and `atom` for the specific atom

atom. Moreover, to make code clearer, many Erlang programmers choose to quote all atoms in type declarations, giving us 'atom' instead of atom. This makes it explicit that 'atom' was meant to be a singleton type, and not a built-in type where the programmer forgot the parentheses.

Following is a table of built-in types provided with the language. Note that they do not all have the same syntax as union types do. Some of them, like binaries and tuples, have a special syntax to make them friendlier to use.

any()	Any Erlang term at all. This is a special type that means that no term or type is valid. Usually, when Dialyzer boils down the possible return values of a function to none(), it means the function should crash. It is synonymous with "this stuff won't work."
none()	
pid()	A process identifier. A port is the underlying representation of file descriptors (which we rarely see unless we go dig deep inside the innards of Erlang libraries), sockets, or generally things that allow Erlang to communicate with the outside world, such as the erlang:open_port/2 function. In the Erlang shell, they look like #Port<0.638>.
port()	
reference()	Unique values returned by make_ref() or erlang:monitor/2.
atom()	Atoms in general.
binary()	A blob of binary data.
<<_:Integer>>	A binary of a known size, where <i>Integer</i> is the size.
<<_:*Integer>>	A binary that has a given unit size, but of unspecified length.
<<_:Integer, _:*OtherInteger>>	A mix of both previous forms to specify that a binary can have a minimum length.
integer()	Any integer. A range of integers. For example, if you wanted to represent a number of months in a year, the range 1..12 could be defined. Note that Dialyzer reserves the right to expand this range into a bigger one.
N..M	
non_neg_integer()	Integers that are greater or equal to 0.
pos_integer()	Integers greater than 0.
neg_integer()	Integers up to -1
float()	Any floating point number.
fun()	Any kind of function.
fun((...)) -> Type	An anonymous function of any arity that returns a given type. A given function that returns lists could be noted as fun((...)) -> list().
fun(() -> Type)	An anonymous function with no arguments, returning a term of a given type.
fun((Type1, Type2,	An anonymous function taking a given number of arguments of a known

<code>..., TypeN) -> Type)</code>	type. An example could be a function that handles an integer and a floating point value, which could be declared as <code>fun((integer(), float()) -> any())</code> .
<code>[]</code>	An empty list.
	A list containing a given type. A list of integers could be defined as <code>[integer()]</code> . Alternatively, it can be written as <code>list(Type())</code> . The type <code>list()</code> is shorthand for <code>[any()]</code> . Lists can sometimes be improper (like <code>[1, 2 a]</code>). As such, Dialyzer has types declared for improper lists with <code>improper_list(TypeList, TypeEnd)</code> . The improper list <code>[1, 2 a]</code> could be typed as <code>improper_list(integer(), atom())</code> , for example. Then, to make matters more complex, it is possible to have lists where we are not actually sure whether the list will be proper or not. In such circumstances, the type <code>maybe_improper_list(TypeList, TypeEnd)</code> can be used.
<code>[Type()]</code>	
<code>[Type(), ...]</code>	This special case of <code>[Type()]</code> mentions that the list can not be empty.
<code>tuple()</code>	Any tuple.
<code>{Type1, Type2, ..., TypeN}</code>	A tuple of a known size, with known types. For example, a binary tree node could be defined as <code>{node, any(), any(), any(), any()}</code> , corresponding to <code>{node, LeftTree, RightTree, Key, Value}</code> .



Given the built-in types above, it becomes a bit easier to imagine how we'd define types for our Erlang programs. Some of it is still missing though. Maybe things are too vague, or not appropriate for our needs. You do remember one of the `discrepN` modules' errors mentioning the type `number()`. That type is neither a singleton type, neither a built-in type. It would then be a union type, which means we could define it ourselves.

The notation to represent the union of types is the pipe (`|`). Basically, this lets us say that a given type `TypeName` is represented as the union of `Type1 | Type2 | ... | TypeN`. As such, the `number()` type, which includes integers and floating point values, could be represented as `integer() | float()`. A boolean value could be defined as `'true' | 'false'`. It is also possible to define types where only one other type is used. Although it looks like a union type, it is in fact an *alias*.

In fact, many such aliases and type unions are pre-defined for you. Here are some of them:

<code>term()</code>	This is equivalent to <code>any()</code> and was added because other tools used <code>term()</code> before. Alternatively, the <code>_</code> variable can be used as an alias of both <code>term()</code> and <code>any()</code> .
<code>boolean()</code>	<code>'true' 'false'</code>
<code>byte()</code>	Defined as <code>0..255</code> , it's any valid byte in existence.

char()	It's defined as <code>0..16#10ffff</code> , but it isn't clear whether this type refers to specific standards for characters or not. It's extremely general in its approach to avoid conflicts.
number()	<code>integer() float()</code>
maybe_improper_list()	This is a quick alias for <code>maybe_improper_list(any(), any())</code> for improper lists in general.
maybe_improper_list(<i>T</i>)	Where <i>T</i> is any given type. This is an alias for <code>maybe_improper_list(<i>T</i>, any())</code> .
string()	A string is defined as <code>[char()]</code> , a list of characters. There is also <code>nonempty_string()</code> , defined as <code>[char(), ...]</code> . Sadly, there is so far no string type for binary strings only, but that's more because they're blobs of data that are to be interpreted in whatever type you choose.
iolist()	Ah, good old iolists. They're defined as <code>maybe_improper_list(char() binary() iolist(), binary() [])</code> . You can see that the iolist is itself defined in terms of iolists. Dialyzer does support recursive types, starting with R13B04. Before then you couldn't use them, and types like iolists could only be defined through some arduous gymnastics.
module()	This is a type standing for module names, and is currently an alias of <code>atom()</code> .
timeout()	<code>non_neg_integer() 'infinity'</code>
node()	An Erlang's node name, which is an atom.
no_return()	This is an alias of <code>none()</code> intended to be used in the return type of functions. It is particularly meant to annotate functions that loop (hopefully) forever, and thus never return.

Well, that makes a few types already. Earlier, I did mention a type for a tree written as `{node, any(), any(), any(), any()}`. Now that we know a bit more about types, we could declare it in a module.

The syntax for type declaration in a module is:

```
-type TypeName() :: TypeDefinition.
```

As such, our tree could have been defined as:

```
-type tree() :: {'node', tree(), tree(), any(), any()}.
```

Or, by using a special syntax that allows to use variable names as type comments:

```
-type tree() :: {'node', Left::tree(), Right::tree(), Key::any(), Value::any()}.
```

But that definition doesn't work, because it doesn't allow for a tree to be empty. A better tree definition can be built by thinking recursively, much like we did with our `tree.erl` module back in [Recursion](#). An empty tree, in that module, is defined as `{node, 'nil'}`. Whenever we hit such a node in a recursive function, we stop. A regular non-empty node is noted as `{node, Key, Val, Left, Right}`. Translating this into a type gives us a tree node of the following form:

```
-type tree() :: {'node', 'nil'}  
    | {'node', Key::any(), Val::any(), Left::tree(), Right::tree()}.
```

That way, we have a tree that is either an empty node or a non-empty node. Note that we could have used 'nil' instead of {'node', 'nil'} and Dialyzer would have been fine with it. I just wanted to respect the way we had written our tree module. There's another piece of Erlang code we might want to give types to but haven't thought of yet...

What about records? They have a somewhat convenient syntax to declare types. To see it, let's imagine a #user{} record. We want to store the user's name, some specific notes (to use our tree() type), the user's age, a list of friends, and some short biography.

```
-record(user, {name = "" :: string(),  
             notes :: tree(),  
             age :: non_neg_integer(),  
             friends = [] :: [#user{}],  
             bio :: string() | binary()}).
```

The general record syntax for type declarations is Field :: Type, and if there's a default value to be given, it becomes Field = Default :: Type. In the record above, we can see that the name needs to be a string, the notes has to be a tree, and the age any integer from 0 to infinity (who knows how old people can get!). An interesting field is friends. The [#user{}] type means that the user records can hold a list of zero or more other user records. It also tells us that a record can be used as a type by writing it as #RecordName{}. The last part tells us that the biography can be either a string or a binary.

Furthermore, to give a more uniform style to type declarations and definitions, people tend to add an alias such as -type Type() :: #Record{}. We could also change the friends definition to use the user() type, ending up as follows:

```
-record(user, {name = "" :: string(),  
             notes :: tree(),  
             age :: non_neg_integer(),  
             friends = [] :: [user()],  
             bio :: string() | binary()}).  
  
-type user() :: #user{}.
```

You'll note that I defined types for all fields of the record, but some of them have no default value. If I were to create a user record instance as #user{age=5}, there would be no type error. All record field definitions have an implicit 'undefined' union added to them if no default value is provided for them. For earlier versions, the declaration would have caused type errors.

Typing Functions

While we could be defining types all day and night, filling files and files with them, then printing the files, framing them and feeling strongly accomplished, they won't automatically

be used by Dialyzer's type inference engine. Dialyzer doesn't work from the types you declare to narrow down what is possible or impossible to execute.

Why the hell would we declare these types then? Documentation? Partially. There is an additional step to making Dialyzer understand the types you declared. We need to pepper type signature declarations over all the functions we want augmented, bridging our type declarations with the functions inside modules.



We have spent most of the chapter on things like 'here is the syntax for this and that', but now it's time to get practical. A simple example of things needing to be typed could be playing cards. There are four suits: spades, clubs, hearts, and diamonds. Cards can then be numbered from 1 to 10 (where the ace is 1), and then be a Jack, Queen, or King.

In regular times, we'd represent cards probably as {suit, CardValue} so that we could have the ace of spades as {spades, 1}. Now, instead of just having this up in the air, we can define types to represent this:

```
-type suit() :: spades | clubs | hearts | diamonds.  
-type value() :: 1..10 | j | q | k.  
-type card() :: {suit(), value()}.
```

The `suit()` type is simply the union of the four atoms that can represent suits. The values can be any card from one to ten (1..10), or j, q, or k for the face cards. The `card()` type joins them together as a tuple.

These three types can now be used to represent cards in regular functions and give us some interesting guarantees. Take the following `cards.erl` module for example:

```
-module(cards).  
-export([kind/1, main/0]).  
  
-type suit() :: spades | clubs | hearts | diamonds.  
-type value() :: 1..10 | j | q | k.  
-type card() :: {suit(), value()}.  
  
kind({_, A}) when A >= 1, A =< 10 -> number;  
kind(_) -> face.  
  
main() ->  
    number = kind({spades, 7}),  
    face = kind({hearts, k}),
```

```
number = kind([{rubies, 4}),
face   = kind([{clubs, q}).
```

The `kind/1` function should return whether a card is a face card or a number card. You will notice that the suit is never checked. In the `main/0` function you can see that the third call is made with the `rubies` suit, something we obviously didn't intend in our types, and likely not in the `kind/1` function:

```
$ erl
...
1> c(cards).
{ok,cards}
2> cards:main().
face
```

Everything works fine. That shouldn't be the case. Even running Dialyzer does nothing. However, if we add the following type signature to the `kind/1` function:

```
-spec kind(card()) -> face | number.
kind({_, A}) when A >= 1, A =< 10 -> number;
kind(_) -> face.
```

Then something more interesting will happen. But before we run Dialyzer, let's see how that works. Type signatures are of the form `-spec FunctionName(ArgumentTypes) -> ReturnTypes..` In the specification above we say that the `kind/1` function accepts cards as arguments, according to the `card()` type we created. It also says the function either returns the atom `face` or `number`.

Running Dialyzer on the module yields the following:

```
$ dialyzer cards.erl
Checking whether the PLT /home/ferd/.dialyzer_plt is up-to-date... yes
Proceeding with analysis...
cards.erl:12: Function main/0 has no local return
cards.erl:15: The call cards:kind({'rubies',4}) breaks the contract (card()) -> 'face' | 'number'
done in 0m0.80s
done (warnings were emitted)
```



Oh bloody fun. Calling `kind/1` with a "card" that has the `rubies` suit isn't a valid thing according to our specifications.

In this case, Dialyzer respects the type signature we gave, and when it analyzes the `main/0` function, it figures out that there is a bad use of `kind/1` in there. This prompts the warning from

line 15 (`number = kind({rubies, 4}),`). Dialyzer from then on assumes that the type signature is reliable, and that if the code is to be used according to it, it would logically not be valid. This breach in the contract propagates to the `main/0` function, but there isn't much that can be said at that level as to why it fails; just that it does.

Note: Dialyzer only complained about this once a type specification was defined. Before a type signature was added, Dialyzer couldn't assume that you planned to use `kind/1` only with `card()` arguments. With the signature in place, it can work with that as its own type definition.

Here's a more interesting function to type, in `convert.erl`:

```
-module(convert).
-export([main/0]).

main() ->
    [_,_] = convert({a,b}),
    {_,_} = convert([a,b]),
    [_,_] = convert([a,b]),
    {_,_} = convert({a,b}).

convert(Tup) when is_tuple(Tup) -> tuple_to_list(Tup);
convert(L = [_|_]) -> list_to_tuple(L).
```

When reading the code, it is obvious to us that the two last calls to `convert/1` will fail. The function accepts a list and returns a tuple, or a tuple and returns a list. If we run Dialyzer on the code though, it'll find nothing.

That's because Dialyzer infers a type signature similar to:

```
-spec convert(list() | tuple()) -> list() | tuple().
```

Or to put it in words, the function accepts lists and tuples, and returns lists in tuples. This is true, but this is sadly a bit too true. The function isn't as permissive as the type signature would imply. This is one of the places where Dialyzer sits back and tries not to say too much without being 100% sure of the problems.

To help Dialyzer a bit, we can send in a fancier type declaration:

```
-spec convert(tuple()) -> list();
            (list()) -> tuple().
convert(Tup) when is_tuple(Tup) -> tuple_to_list(Tup);
convert(L = [_|_]) -> list_to_tuple(L).
```

Rather than putting `tuple()` and `list()` types together into a single union, this syntax allows you to declare type signatures with alternative clauses. If you call `convert/1` with a tuple, we expect a list, and the opposite in the other case.

With this more specific information, Dialyzer can now give more interesting results:

```
$ dialyzer convert.erl
Checking whether the PLT /home/ferd/.dialyzer_plt is up-to-date... yes
Proceeding with analysis...
convert.erl:4: Function main/0 has no local return
convert.erl:7: The pattern [_,_] can never match the type tuple()
done in 0m0.90s
done (warnings were emitted)
```

Ah, there it finds the error. Success! We can now use Dialyzer to tell us what we knew. Of course putting it that way sounds useless, but when you type your functions right and make a tiny mistake that you forget to check, Dialyzer will have your back, which is definitely better than an error logging system waking you up at night (or having your car keyed by your operations guy).

Note: some people will prefer the following syntax for multi-clause type signature:

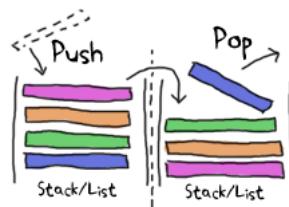
```
-spec convert(tuple()) -> list()
;    (list()) -> tuple().
```

which is exactly the same, but puts the semi-colon on another line because it might be more readable. There is no widely accepted standard at the time of this writing.

By using type definitions and specifications in that way, we're in fact able to let Dialyzer find errors with our earlier `discrep` modules. See how `discrep4.erl` does it.

Typing Practice

I've been writing a queue module, for First In, First Out (FIFO) operations. You should know what queues are, given Erlang's mailboxes are queues. The first element added will be the first one to be popped (unless we do selective receives). The module works as explained in this image we've seen a few times already:



To simulate a queue, we use two lists as stacks. One list stores the new elements and one list lets us remove them from the queue. We always add to the same list, and remove from the second one. When the list we remove from is empty, we reverse the list we add items to and it becomes the new list to remove from. This generally guarantees better average performance than using a single list to do both tasks.

Here's my module, with a few type signatures I added to check it with Dialyzer:

```
-module(fifo_types).
-export([new/0, push/2, pop/1, empty/1]).
```

```

-export([test/0]).

-spec new() -> {fifo, [], []}.
new() -> {fifo, [], []}.

-spec push({fifo, In:list(), Out:list()}, term()) -> {fifo, list(), list()}.
push({fifo, In, Out}, X) -> {fifo, [X|In], Out}.

-spec pop({fifo, In:list(), Out:list()}) -> {term(), {fifo, list(), list()}}.
pop({fifo, [], []}) -> erlang:error('empty fifo');
pop({fifo, In, []}) -> pop({fifo, [], lists:reverse(In)});
pop({fifo, In, [H|T]}) -> {H, {fifo, In, T}}.

-spec empty({fifo, [], []}) -> true;
empty({fifo, list(), list()}) -> false.
empty({fifo, [], []}) -> true;
empty({fifo, _, _}) -> false.

test() ->
    N = new(),
    {2, N2} = pop(push(push(new(), 2), 5)),
    {5, N3} = pop(N2),
    N = N3,
    true = empty(N3),
    false = empty(N2),
    pop({fifo, [a|b], [e]}).

```

I defined a queue as a tuple of the form `{fifo, list(), list()}`. You'll notice I didn't define a `fifo()` type, mostly because I simply wanted to be able to easily make different clauses for empty queues and filled queues. The `empty(...)` type specification reflects that.

Note: You will notice that in the function `pop/1` that I do not specify the `none()` type even though one of the function clauses calls `erlang:error/1`.

The type `none()`, as mentioned earlier, is a type that means a given function will not return. If the function might either fail or return a value, it is useless to type it as returning both a value and `none()`. The `none()` type is always assumed to be there, and as such, the union `Type() | none()` is the same as `Type()` alone.

The circumstances where `none()` is warranted is whenever you're writing a function that always fails when called, such as if you were implementing `erlang:error/1` yourself.

Now all the type specifications above do appear to make sense. Just to make sure, during code reviewing, I ask you to run Dialyzer with me to see the results:

```

$ dialyzer fifo_types.erl
Checking whether the PLT /home/ferd/.dialyzer_plt is up-to-date... yes
Proceeding with analysis...
fifo_types.erl:16: Overloaded contract has overlapping domains; such contracts are currently unsupported and are si
fifo_types.erl:21: Function test/0 has no local return

```

```
fifo_types.erl:28: The call fifo_types:pop({'fifo',nonempty_improper_list('a','b'),['e','...']}) breaks the contract ({'fifo',in:[a|done in 0m0.96s  
done (warnings were emitted)}
```

Silly me. We've got a bunch of errors showing up. And curses, they're not so easy to read. The second one, 'Function test/0 has no local return', is at least something we know how to handle — we will just skip to the next one and it should disappear.

For now let's focus on the first one, the one about contracts with overlapping domains. If we go into fifo_types on line 16, we see this:

```
-spec empty({fifo, [], []}) -> true;  
      ({fifo, list(), list()}) -> false.  
empty({fifo, [], []}) -> true;  
empty({fifo, _, _}) -> false.
```

So what are said overlapping domains? We have to refer to the mathematical concepts of domain and image. To put it simply, the domain is the set of all possible input values to a function, and the image is the set of all possible output values of a function. Overlapping domain thus refer to two sets of input that overlap.



To find the source of the problem we have to look at list(). If you remember the large tables from earlier, list() is pretty much the same as [any()]. And you'll also remember that both of these types both also include empty lists. And there's your overlapping domain. When list() is specified as a type, it overlaps with []. To fix this, we need to replace the type signature as follows:

```
-spec empty({fifo, [], []}) -> true;  
      ({fifo, nonempty_list(), nonempty_list()}) -> false.
```

or alternatively:

```
-spec empty({fifo, [], []}) -> true;  
      ({fifo, [any(), ...], [any(), ...]}) -> false.
```

Then running Dialyzer again will get rid of the warning. However, this is not enough. What if someone sent in {fifo, [a,b], []}? Even if Dialyzer might not complain about it, it is somewhat obvious for human readers that the type specification above doesn't take this into account. The spec doesn't match the intended use of the function. We can instead give more details and take the following approach:

```
-spec empty({fifo, [], []}) -> true;  
      ({fifo, [any(), ...], []}) -> false;
```

```
({fifo, [], [any(), ...]}) -> false;  
({fifo, [any(), ...], [any(), ...]}) -> false.
```

Which will both work, and have the right meaning.

On to the next error (which I broke into multiple lines):

```
fifo_types.erl:28:  
The call fifo_types:pop({'fifo',nonempty_improper_list('a','b'),['e','...']})  
breaks the contract  
({fifo,'In':[any()],'Out':[any()]}) -> {term(),'fifo',[any()][any()]}  
}
```

Translated into human, this means that on line 28, there's a call to `pop/1` that has inferred types breaking the one I specified in the file:

```
pop({fifo, [a|b], [e]}).
```

That's the call. Now, the error message says that it identified an improper list (that happens to not be empty), which is entirely right; `[a|b]` is an improper list. It also mentions that it breaks a contract. We need to match the type definition that is broken between the following, coming from the error message:

```
{'fifo',nonempty_improper_list('a','b'),['e','...']}  
'fifo',{In:[any()],Out:[any()]}  
{term(),'fifo',[any()][any()]}
```

The issue can be explained in one of three ways:

1. The type signatures are right, the call is right and the problem is the return value expected.
2. The type signatures are right, the call is wrong and the problem is the input value given.
3. The call is right, but the type signatures are wrong.

We can eliminate the first one right away. We're not actually doing anything with the return value. This leaves the second and third option. The decision boils down to whether we wanted improper lists to be used with our queues or not. This is a judgment call to be made by the writer of the library, and I can say without a doubt that I didn't intend improper lists to be used with this code. In fact you very rarely want improper lists. The winner is number 2, the call is wrong. To solve it, drop the call or fix it:

```
test() ->  
    N = new(),  
    {2, N2} = pop(push(push(new(), 2), 5)),  
    ...  
    pop({fifo, [a, b], [e]}).
```

And running Dialyzer again:

```
$ dialyzer fifo_types.erl  
Checking whether the PLT /home/ferd/.dialyzer_plt is up-to-date... yes
```

```
Proceeding with analysis... done in 0m0.90s
done (passed successfully)
```

That now makes more sense.

Exporting Types

That is all very well. We have types, we have signatures, we have additional safety and verifications. So what would happen if we wanted to use our queue in another module? What about any other module we frequently use, things like dict, gb_trees, or ETS tables? How can we use Dialyzer to find type errors related to them?

We can use types coming from other modules. Doing so usually requires rummaging through documentation to find them. For example, the ets module's documentation has the following entries:

```
---
DATA TYPES

continuation()
  Opaque continuation used by select/1 and select/3.

match_spec() = [{match_pattern(), [term()], [term()]}]
  A match specification, see above.

match_pattern() = atom() | tuple()

tab() = atom() | tid()

tid()
  A table identifier, as returned by new/2.
---
```

Those are the data types exported by ets. If I had a type specification that were to accept ETS tables, a key, and returns a matching entry I could define it maybe like this:

```
-spec match(ets:tab(), Key::any()) -> Entry::any().
```

And that's about it.

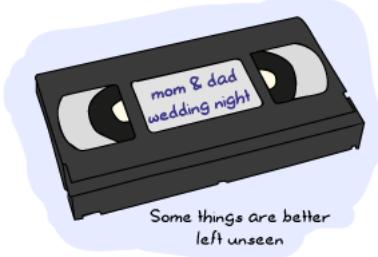
Exporting our own types works pretty much the same as we do for functions. All we need to do is add a module attribute of the form `-export_type([TypeName/Arity])`. For example, we could have exported the `card()` type from our `cards` module by adding the following line:

```
-module(cards).
-export([kind/1, main/0]).  
  
-type suit() :: spades | clubs | hearts | diamonds.
-type value() :: 1..10 | j | q | k.
-type card() :: {suit(), value()}.
```

```
-export_type([card/0]).
```

```
...
```

And from then on, if the module is visible to Dialyzer (either by adding it to the PLT or analyzing it at the same time as any other module), you can reference it from any other bit of code as `cards:card()` in type specifications.



Doing this will have one downside, though. Using a type like this doesn't forbid anyone using the card module from ripping the types apart and toying with them. Anyone could be writing pieces of code that matches on the cards, a bit like `{suit, _} = ...`. This isn't always a good idea: it keeps us from being able to change the implementation of the `cards` module in the future. This is something we'd especially like to enforce in modules that represent data structures, such as `dict` or `fifo_types` (if it exported types).

Dialyzer allows you to export types in a way that tells your users "you know what? I'm fine with you using my types, but don't you dare look inside of them!". It's a question of replacing a declaration of the kind:

```
-type fifo() :: {fifo, list(), list()}.
```

by:

```
-opaque fifo() :: {fifo, list(), list()}.
```

Then you can still export it as `-export_type([fifo/0])`.

Declaring a type as *opaque* means that only the module that defined the type has the right to look at how it's made and make modifications to it. It forbids other modules from pattern matching on the values other than the whole thing, guaranteeing (if they use Dialyzer) that they will never be bit by a sudden change of implementation.

Don't Drink Too Much Kool-Aid:

Sometimes the implementation of opaque data types is either not strong enough to do what it should or is actually problematic (i.e. buggy). Dialyzer does not take the spec of a function into account until it has first inferred the success typing for the function.

This means that when your type looks rather generic without any -type information taken into account, Dialyzer might get confused by some opaque types. For example, Dialyzer

analyzing an opaque version of the `card()` data type might see it as `{atom(), any()}` after inference. Modules using `card()` correctly might see Dialyzer complaining because they're breaking a type contract even if they aren't. This is because the `card()` type itself doesn't contain enough information for Dialyzer to connect the dots and realize what's really going on.

Usually, if you see errors of that kind, tagging your tuple helps. Moving from a type of the form `-opaque card() :: {suit(), value()}.` to `-opaque card() :: {card, suit(), value()}.` might get Dialyzer to work fine with the opaque type.

The Dialyzer implementers are currently trying to make the implementation of opaque data types better and strengthen their inference. They're also trying to make user-provided specs more important and to trust them better during Dialyzer's analysis, but this is still a work in progress.

Typed Behaviours

Back in [Clients and Servers](#), we've seen that we could declare behaviours using the `behaviour_info/1` function. The module exporting this function would give its name to the behaviour, and a second module could implement callbacks by adding `-behaviour(ModName).` as a module attribute.

The behaviour definition of the `gen_server` module, for example, is:

```
behaviour_info(callbacks) ->
  [{init, 1}, {handle_call, 3}, {handle_cast, 2}, {handle_info, 2},
   {terminate, 2}, {code_change, 3}];
behaviour_info(_Other) ->
  undefined.
```

The problem with that one is that there is no way for Dialyzer to check type definitions for that. In fact, there is no way for the behaviour module to specify what kind of types it expects the callback modules to implement, and there's thus no way for Dialyzer to do something about it.

Starting with R15B, The Erlang/OTP compiler was upgraded so that it now handles a new module attribute, named `-callback`. The `-callback` module attribute has a similar syntax to spec. When you specify function types with it, the `behaviour_info/1` function gets automatically declared for you, and the specifications get added to the module metadata in a way that lets Dialyzer do its work. For example, here's the declaration of the `gen_server` starting with R15B:

```
-callback init(Args :: term()) ->
  {ok, State :: term()} | {ok, State :: term(), timeout() | hibernate} |
  {stop, Reason :: term()} | ignore.
-callback handle_call(Request :: term(), From :: {pid(), Tag :: term()},
                      State :: term()) ->
  {reply, Reply :: term(), NewState :: term()} |
```

```

{reply, Reply :: term(), NewState :: term(), timeout() | hibernate} |
{noreply, NewState :: term()} |
{noreply, NewState :: term(), timeout() | hibernate} |
{stop, Reason :: term(), Reply :: term(), NewState :: term()} |
{stop, Reason :: term(), NewState :: term()}.

-callback handle_cast(Request :: term(), State :: term()) ->
{noreply, NewState :: term()} |
{noreply, NewState :: term(), timeout() | hibernate} |
{stop, Reason :: term(), NewState :: term()}.

-callback handle_info(Info :: timeout() | term(), State :: term()) ->
{noreply, NewState :: term()} |
{noreply, NewState :: term(), timeout() | hibernate} |
{stop, Reason :: term(), NewState :: term()}.

-callback terminate(Reason :: (normal | shutdown | {shutdown, term()} | term()),
State :: term()) ->
term().

-callback code_change(OldVsn :: (term() | {down, term()}), State :: term(),
Extra :: term()) ->
{ok, NewState :: term()} | {error, Reason :: term()}.

```

And none of your code should break from the behaviour changing things. Do realize, however, that a module cannot use both the `-callback` form and the `behaviour_info/1` function at once. Only one or the other. This means if you want to create custom behaviours, there is a rift between what can be used in versions of Erlang prior to R15, and what can be used in latter versions.

The upside is that newer modules will have Dialyzer able to do some analysis to check for errors on the types of whatever is returned there to help.

Polymorphic Types

Oh boy, what a section title. If you've never heard of *polymorphic types* (or alternatively, *parameterized types*), this might sound a bit scary. It's fortunately not as complex as the name would let us believe.



The need for polymorphic types comes from the fact that when we're typing different data structures, we might sometimes find ourselves wanting to be pretty specific about what they can store. We might want our queue from earlier in the chapter to sometimes handle anything, sometimes handle only playing cards, or sometimes handle only integers. In these two last cases, the issue is that we might want Dialyzer to be able to complain that we're

trying to put floating point numbers in our integer queue, or tarot cards in our playing cards queue.

This is something impossible to enforce by strictly doing types the way we were doing them. Enter polymorphic types. A polymorphic type is a type that can be 'configured' with other types. Luckily for us, we already know the syntax to do it. When I said we could define a list of integers as [integer()] or list(integer()), those were polymorphic types. It's a type that accepts a type as an argument.

To make our queue accept only integers or cards, we could have defined its type as:

```
-type queue(Type) :: {fifo, list(Type), list(Type)}.
-export_type([queue/1]).
```

When another module wishes to make use of the fifo/1 type, it needs to parameterize it. So a new deck of cards in the cards module could have had the following signature:

```
-spec new() -> fifo:queue(card()).
```

And Dialyzer, if possible, would have tried to analyze the module to make sure that it only submits and expects cards from the queue it handles.

For a demonstration, we decided to buy a zoo to congratulate ourselves on being nearly done with Learn You Some Erlang. In our zoo, we have two animals: a red panda and a squid. Granted, it is a rather modest zoo, although that shouldn't keep us from setting the entry fee sky high.

We've decided to automate the feeding of our animals, because we're programmers, and programmers sometimes like to automate stuff out of laziness. After doing a bit of research, we've found that red pandas can eat bamboo, some birds, eggs, and berries. We've also found that squids can fight with sperm whales, so we decided to feed them just that with our zoo.erl module:

```
-module(zoo).
-export([main/0]).

feeder(red_panda) ->
    fun() ->
        element(random:uniform(4), {bamboo, birds, eggs, berries})
    end;
feeder(squid) ->
    fun() -> sperm_whale end.

feed_red_panda(Generator) ->
    Food = Generator(),
    io:format("feeding ~p to the red panda~n", [Food]),
    Food.

feed_squid(Generator) ->
```

```

Food = Generator(),
io:format("throwing ~p in the squid's aquarium~n", [Food]),
Food.

main() ->
    %% Random seeding
    <<A:32, B:32, C:32>> = crypto:rand_bytes(12),
    random:seed(A, B, C),
    %% The zoo buys a feeder for both the red panda and squid
    FeederRP = feeder(red_panda),
    FeederSquid = feeder(squid),
    %% Time to feed them!
    %% This should not be right!
    feed_squid(FeederRP),
    feed_red_panda(FeederSquid).

```

This code makes use of `feeder/1`, which takes an animal name and returns a feeder (a function that returns food items). Feeding the red panda should be done with a red panda feeder, and feeding the squid should be done with a squid feeder. With function definitions such as `feed_red_panda/1` and `feed_squid/1`, there is no way to be alerted by the wrong use of a feeder. Even with run time checks, it's impossible to do. As soon as we serve food, it's too late:

```

1> zoo:main().
throwing bamboo in the squid's aquarium
feeding sperm_whale to the red panda
sperm_whale

```

Oh no, our little animals are not meant to eat that way! Maybe types can help. The following type specifications could be devised to help us, using the power of polymorphic types:

```

-type red_panda() :: bamboo | birds | eggs | berries.
-type squid() :: sperm_whale.
-type food(A) :: fun(() -> A).

-spec feeder(red_panda) -> food(red_panda());
    (squid) -> food(squid()).
-spec feed_red_panda(food(red_panda())) -> red_panda().
-spec feed_squid(food(squid())) -> squid().

```

The `food(A)` type is the one of interest here. `A` is a free type, to be decided upon later. We then qualify the food type in `feeder/1`'s type specification by doing `food(red_panda())` and `food(squid())`. The food type is then seen as `fun(() -> red_panda())` and `fun(() -> squid())` instead of some abstract function returning something unknown. If you add these specs to the file and then run Dialyzer on it, the following happens:

```

$ dialyzer zoo.erl
Checking whether the PLT /Users/ferd/.dialyzer_plt is up-to-date... yes
Proceeding with analysis...
zoo.erl:18: Function feed_red_panda/1 will never be called
zoo.erl:23: The contract zoo:feed_squid(food(squid())) -> squid() cannot be right because the inferred return for feec

```

```
zoo.erl:29: Function main/0 has no local return
done in 0m0.68s
done (warnings were emitted)
```

And the error is right. Hooray for polymorphic types!

While the above is pretty useful, minor changes in your code can have unexpected consequences in what Dialyzer is able to find. For example, if the `main/0` function had the following code:

```
main() ->
    %% Random seeding
    <<A:32, B:32, C:32>> = crypto:rand_bytes(12),
    random:seed(A, B, C),
    %% The zoo buys a feeder for both the red panda and squid
    FeederRP = feeder(red_panda),
    FeederSquid = feeder(squid),
    %% Time to feed them!
    feed_squid(FeederSquid),
    feed_red_panda(FeederRP),
    %% This should not be right!
    feed_squid(FeederRP),
    feed_red_panda(FeederSquid).
```

Things would not be the same. Before the functions are called with the wrong kind of feeder, they're first called with the right kind. As of R15B01, Dialyzer would not find an error with this code. This is because Dialyzer does not necessarily keep information regarding whether the anonymous function is being called at all in the feeding functions when some complex module-local refinement is being done.

Even if this is a bit sad for many static typing fans, we have been thoroughly warned. The following quote comes from the paper describing the implementation of success typing for Dialyzer:

A success typing is a type signature that over-approximates the set of types for which the function can evaluate to a value. The domain of the signature includes all possible values that the function could accept as parameters, and its range includes all possible return values for this domain.

However weak this might seem to aficionados of static typing, success typings have the property that they capture the fact that if the function is used in a way not allowed by its success typing (e.g., by applying the function with parameters p / α) this application will definitely fail. This is precisely the property that a defect detection tool which never "cries wolf" needs. Also, success typings can be used for automatic program documentation because they will never fail to capture some possible — no matter how unintended — use of a function.

Again, keeping in mind that Dialyzer is optimistic in its approach is vital to working efficiently with it.

If this still depresses you too much, you can try adding the `-Woverspecs` option to Dialyzer:

```
$ dialyzer zoo.erl -Woverspecs
Checking whether the PLT /home/ferd/.dialyzer_plt is up-to-date... yes
Proceeding with analysis...
zoo.erl:17: Type specification zoo:feed_red_panda(food(red_panda())) -> red_panda() is a subtype of the success
done in 0m0.94s
done (warnings were emitted)
```

This warns you that in fact, your specification is way too strict for what your code is expected to accept, and tells you (albeit indirectly) that you should either make your type specification looser, or validate your inputs and outputs better in your functions to reflect the type specification.



You're my Type

Dialyzer will often prove to be a true friend when programming Erlang, although the frequent nags might tempt you to just drop it. One thing to remember is that Dialyzer is practically never wrong, and you will likely be. You might feel like some errors mean nothing, but contrary to many type systems, Dialyzer only speaks out when it knows it's right, and bugs in its code base are rare. Dialyzer might frustrate you, force you to be humble, but it will be very unlikely to be the source of bad, unclean code.

Note: While writing this chapter, I ended up having some nasty Dialyzer error message when working with a more complete version of the stream module. I was annoyed enough to go complain on IRC about it, how Dialyzer was not good enough to handle my complex use of types.

Silly me. It turns out (unsurprisingly so) that *I* was wrong, and Dialyzer was right, all along. It would keep telling me my -spec was wrong, and I kept believing it wasn't. I lost my fight, Dialyzer and my code won. This is a good thing, I believe.

So hey, that's about it for Learn You Some Erlang for great good! Thanks for reading it. There's not much more to say, but if you feel like getting a list of more topics to explore and some general words from me, you can go read the guide's conclusion. Godspeed! You Concurrent Emperor.

Conclusion

A Few Words

I see you chose to read the conclusion after all. Good for you. Before I get to point you to a bunch of interesting topics that you could explore, were you to pick Erlang as a development language you want to learn more about, I'd like to take a few lines to say writing Learn You Some Erlang has been one hell of a ride. It took me three years of hard work while studying and working full time, and juggling every day's life needs (if I had children, they'd have died of neglect a while ago now).

This site, coupled with some luck and some more work, allowed me to get jobs, both as an Erlang trainer, course material writer, and developer. It allowed me to travel around the world and meet a crapload of interesting people. It drained a lot of energy, cost me a decent chunk of money and time to run, but it paid back tenfold in most ways imaginable.

I have to give a lot of thanks to the Erlang community in general. They helped me learn stuff, reviewed pages and pages of material for free, fixed my typos, helped me get better at writing English and writing in General. There's been dozen of people helping in many ways. The biggest contributors in terms of time, advice, and general resources are all in this site's FAQ. If you've ever written me an e-mail

telling me you'd buy me a beer, buy it to one of these guys instead; they deserve it as their participation was way more thankless than mine.

The Erlang community as a whole has been very welcoming to the work I've been doing with LYSE, helped make it known to readers (it's even on the official Erlang documentation and website!). Without the concerted efforts of everyone around Erlang, this site would probably have died after four or five chapters, left to be yet another useless website clogging the Internet's tubes. So, hey, thanks.

Other Topics

There's only so many topics I could cover without going over the top. This site, if it were to be turned in dead tree form, would probably yield around 600 pages now. It's taken three years to bring it there, and I'm tired and glad it's over (what am I gonna do with all that free time, now?), but there are still plenty of topics I would have *loved* to cover. Here's a quick list:

Tracing BIFs and DBG

The Erlang VM is traceable inside and out. Got a bug or some stack trace you can't make sense of? Turn on a few trace flags and the VM opens up to you. DBG takes these BIFs and builds an app on top of them. Messages, function calls, function returns, garbage collections, process spawning and dying, etc. It's all traceable and observable. It also tends to

work much better than any debugger for a concurrent language like Erlang. The best about it? It's traceable within Erlang, so you can make Erlang programs that trace themselves! If you look into them and find them a bit hard to digest, you might be okay staying with the `sys` module's tracing functions. They work only on OTP behaviourised processes, but they're often good enough to get going.

Profiling

Erlang comes with a good bunch of different profiling tools to analyze your programs and find all kinds of bottlenecks. The `fprof` and `erprof` tools can be used for time profiling, `cprof` for function calls, `lcnt` for locks, `percept` for concurrency, and `cover` for code coverage. Most of them are built using the tracing BIFs of the language, funnily enough.

More Introspection

Top-like tools exist for Erlang, such as `pman` and `etop`. You can also use the Erlang debugger, but I do recommend `DBG` instead of that one. To explore entire supervision trees for your nodes, `appmon` is your app.

Documentation

`EDoc` is a tool that lets you turn your Erlang modules into HTML documentation. It supports annotations and ways to declare specific pages that allow you to build small websites

to document your code. It's similar to Javadoc, if you've heard of it.

GUIs

The Wx application is the new standard for multiplatform GUI writing with Erlang. I'm terrible at GUI stuff, so it's probably better for everyone I actually didn't cover that one.

Other Useful Libraries

There are plenty of nice libraries coming by default with Erlang not mentioned here. Cryptography tools, web servers, web clients, all kinds of protocol implementations, and so on. You can get a general list of them at <http://www.erlang.org/doc/applications.html>.

Community libraries

There is a crapload of them. I didn't want to cover them because they can tend to change and I didn't want to favor one over the other, but here's a quick list: Rebar3 and erlang.mk if you want build systems, redbug or recon for a friendlier approach to tracing, gproc for a very powerful and flexible process registry, mochiweb, cowboy and yaws if you need web servers, riak_core for a very powerful distribution library for Erlang, hackney as a web client, PropEr, Quickcheck and Triq for kick-ass property-based testing tools (you need to try one of them), entop for a top-like tool, a billion JSON libraries (jsx, jiffy, etc.), UX for Unicode handling and common

algorithm pending R16B, Seresye and exat for some AI tools, database client libraries, lager as very robust logging system that binds itself to Erlang's error_logger, poolboy for some generic message-based pool, and a whole lot more stuff. Community libraries could easily get their own book.

I heard LYSE is also a book?

You heard right. Thanks to No Starch Press, Learn You Some Erlang is available both as a dead tree book and an ebook! At a large 600 black and white pages, including images (in color for ebook copies), you can now have the largest Erlang-themed paperweight and bookcase decoration printed to date (as far as I know). This should ease the sharp pain of reading hundreds of pages on a computer screen.

Your ideas are intriguing to me and I wish to subscribe to your newsletter

I have a blog at ferd.ca where I discuss all kinds of stuff (or at least I want to), but inevitably come back to Erlang topics, due to using it all the time.

Is that it?

Yes, it is. Have a nice day!

Postscript: Maps

About This Chapter

Oh hi. It's been a while. This is an add-on chapter, one that isn't yet part of the printed version of Learn You Some Erlang for great good! Why does this exist? Was the printed book a huge steaming pile of bad information? I hope not (though you can consult the errata to see how wrong it is).

The reason for this chapter is that the Erlang zoo of data types has grown just a bit larger with the R17 release, and reworking the entire thing to incorporate the new type would be a lot of trouble. The R17 release had a lot of changes baked in, and is one of the biggest one in a few years. In fact, the R17 release should be called the 17.0 release, as Erlang/OTP then started changing their R<Major>B<Minor> versioning scheme to a different versioning scheme.

In any case, as with most updates to the language that happened before, I've added notes and a few bits of content here and there through the website so that everything stays up to date. The new data type added, however, deserves an entire chapter on its own.



EEP, EEP!

The road that led to maps being added to the language has been long and tortuous. For years, people have been complaining about records and key/value data structures on multiple fronts in the Erlang community:

- People don't want to put the name of the record every time when accessing it and want dynamic access
- There is no good canonical arbitrary key/value store to target and use in pattern matches
- Something faster than dicts and gb_trees would be nice
- Converting to and from key/value stores is difficult to do well, and native data types would help with this
- Pattern matching is a big one, again!
- Records are a preprocessor hack and problematic at run time (they are tuples)
- Upgrades with records are painful
- Records are bound to a single module
- And so much more!

There ended up being two competing proposals. The first is Richard O'Keefe's frames, a really deeply thought out proposal on how to replace records, and Joe Armstrong's Structs (O'Keefe's proposal describes them briefly and compares them to frames).

Later on, the OTP team came up with Erlang Enhancement Proposal 43 (EEP-43), their own proposal for a "similar" data type. However, much like with community complaints, the two proposals were working on entirely different issues: maps are intended to be very flexible hash maps to replace dicts and gb_trees, and frames are intended to replace records. Maps would also ideally be able to replace records in some use cases, but not all of them while preserving the positive characteristics of records (as we'll see in [Mexican Standoff](#)).

Maps were the one picked to be implemented by the OTP team, and frames are still hanging with their own proposal, somewhere.

The proposal for maps is comprehensive, covers many cases, and highlights many of the design issues in trying to have them fit the language, and for these reasons, their implementation will be gradual. The specification is described in [What Maps Shall Be](#), while the temporary implementation is described in [Stubby Legs for Early Releases](#).

What Maps Shall Be

The Module

Maps are a data type similar to the dict data structure in intent, and has been given a module with a similar interface and semantics. The following operations are supported:

- `maps:new()`: returns a new, empty map.
- `maps:put(Key, Val, Map)`: adds an entry to a map.
- `maps:update(Key, Val, Map)`: returns a map with Key's entry modified or raises the error `badarg` if it's not there.
- `maps:get(Key, Map)` and `maps:find(Key, Map)`: find items in a map. The former returns the value if present or raises the `badkey` error otherwise, whereas the latter returns `{ok, Val}` or `error`. Another version of the function is `maps:get(Key, Map, Default)`, which allows to specify the value to return if the key is missing.
- `maps:remove(Key, Map)`: returns a map with the given element deleted. Similarly, `maps:without([Keys], Map)` will delete multiple elements from the returned map. If a key to be deleted isn't present, the functions behave as if they had deleted it — they return a map without it. The opposite of `maps:without/2` is `maps:with/2`.

It also contains the good old functional standards such as `fold/3` and `map/2`, which work similarly to the functions in the `lists` module. Then there is the set of utility functions such as `size/1`, `is_map/1` (also a guard!), `from_list/1` and `to_list/1`, `keys/1` and `values/1`, and set functions like `is_key/2` and `merge/2` to test for membership and fuse maps together.

That's not all that exciting, and users want more!

The Syntax

Despite the promised gains in speed, the most anticipated aspect of maps is the native syntax they have. Here are the different operations compared to their equivalent module call:

Maps Module Maps Syntax

```
maps:new/1      #{}
maps:put/3      Map#{Key => Val}
maps:update/3   Map#{Key := Val}
maps:get/2      Map#{Key}
maps:find/2     #{Key := Val} = Map
```

Bear in mind that [not all of this syntax has been implemented yet](#). Fortunately for map users, the pattern matching options go wider than this. It is possible to match more than one item out of a map at once:

```
1> Pets = #{ "dog" => "winston", "fish" => "mrs.blub"}.
#{ "dog" => "winston", "fish" => "mrs.blub"}
2> #{ "fish" := CatName, "dog" := DogName} = Pets.
#{ "dog" => "winston", "fish" => "mrs.blub"}
```

Here it's possible to grab the contents of any number of items at a time, regardless of order of keys. You'll note that elements are set with `=>` and matched with `:=`. The `:=` operator can also be used to update an *existing* key in a map:

```
3> Pets#{ "dog" := "chester"}.
#{ "dog" => "chester", "fish" => "mrs.blub"}
4> Pets#{dog := "chester"}.
** exception error: bad argument
   in function  maps:update/3
      called as maps:update(dog,"chester",#{ "dog" => "winston", "fish" => "mrs.blub"})
   in call from erl_eval:'-expr/5-fun-0-/2 (erl_eval.erl, line 257)
   in call from lists:foldl/3 (lists.erl, line 1248)
```

There's more matching in the specification, although it's not available in 17.0 yet:

```
5> #{"favorite" := Animal, Animal := Name} = Pets#{"favorite" := "dog"}.
#{"dog" => "winston", "favorite" => "dog", "fish" => "mrs.blub"}
6> Name.
"winston"
```

Within the same pattern, a known key's value can be used to define a variable (*Animal*) usable as another key, and then use that other key to match the desired value (*Name*). The limitation with this kind of pattern is that there cannot be cycles. For example, matching $\# \{x := y, y := x\} = \text{Map}$ cannot be done due to needing to know *Y* to match *X*, and needing to know *X* to bind *Y*. You also cannot do matching of a key by value ($\# \{x := \text{val}\} = \text{Map}$) because there could be multiple keys with the same value.

Note: The syntax for accessing a single value ($\text{Map}\#\{\text{Key}\}$) is documented as such in the EEP, but is subject to change in the future once implemented, and might be dropped entirely in favor of different solutions.

There's something interesting but unrelated that was added with maps. If you remember in [Starting Out For Real](#), we introduced list comprehensions:

```
7> Weather = [{toronto, rain}, {montreal, storms}, {london, fog},
7>              {paris, sun}, {boston, fog}, {vancouver, snow}].
[{:toronto,rain},
{:montreal,storms},
{:london,fog},
{:paris,sun},
{:boston,fog},
{:vancouver,snow}]
8> FoggyPlaces = [X || {X, fog} <- Weather].
[{:london,boston}]
```

The same kind of stuff can be done with map comprehensions, once they're made available:

```
9> Weather = #{toronto => rain, montreal => storms, london => fog,
9>           paris => sun, boston => fog, vancouver => snow}.
#:{boston => fog,
london => fog,
montreal => storms,
paris => sun,
toronto => rain,
vancouver => snow}
10> FoggyPlaces = [X || X := fog <- Weather].
[London,boston]
```

Here, `X := fog <- Weather` represents a map generator, of the form `Key := Val <- Map`. This can be composed and substituted the same way list generators and binary generators can. Map comprehensions can also generate new maps:

```
11> #{X => foggy || X <- [london,boston]}.
#{boston => foggy, london => foggy}
```

Or to implement the map operation from the `maps` module itself:

```
map(F, Map) ->
#{K => F(V) || K := V <- Map}.
```

And that's most of it! It's extremely refreshing to see this new data type joining the Erlang zoo, and hopefully users will appreciate it.



The Gritty Details

The details aren't *that* gritty, just a little bit. The inclusion of maps in the language will impact a few things. EEP-43 goes into detail to define potential changes, many somewhat still in the air, for parts such as the distributed Erlang protocol, operator precedence, backwards compatibility, and suggestions to Dialyzer extensions (it is yet to see if the support will go as far as the EEP recommends).

Many of the changes have been proposed such that the user doesn't need to think very hard about them. One that is unavoidable, however, is sorting. Previously in the book, the sorting order was defined as:

number < atom < reference < fun < port < pid < tuple < list < bit string

Maps now fit in here:

number < atom < reference < fun < port < pid < tuple < map < list < bit string

Bigger than tuples and smaller than lists. Interestingly enough, maps can be compared to each other based on their keys and values:

```
2> lists:sort([{#1 => 2, 3 => 4}, #{2 => 1}, #{2 => 0, 1 => 4}]).  
[#{2 => 1},#{1 => 4,2 => 0},#{1 => 2,3 => 4}]
```

The sorting is done similarly to lists and tuples: first by size, then by the elements contained. In the case of maps, these elements are in the sorted order of keys, and breaking ties with the values themselves.

Don't Drink Too Much Kool-Aid:

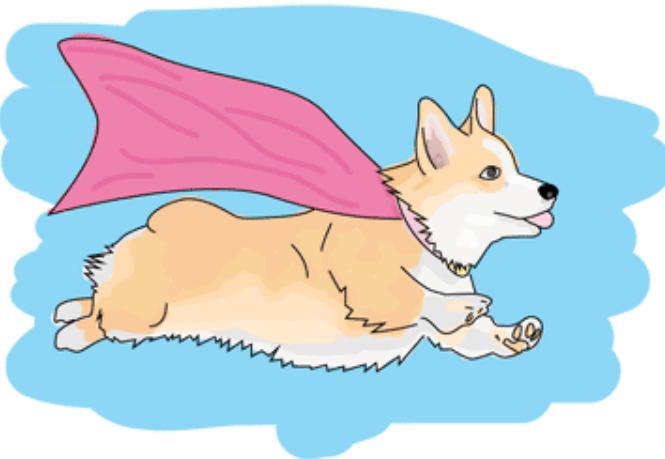
You may notice that while we can't update the key 1.0 of a map with the key 1, it is possible to have them both compare as equal that way! This is one of the long-standing warts of Erlang. While it's very convenient to have all numbers compare equal from time to time, for example, when sorting lists so that 1.0 isn't greater than 9121, this creates confusing expectations when it comes to pattern matching.

For example, while we can expect 1 to be equal to 1.0 (although not *strictly equal*, as with `=:=`), we can't expect to pattern match by doing `1 = 1.0`.

In the case of maps, this means that `Map1 == Map2` isn't a synonym of `Map1 = Map2`. Because Erlang maps respect Erlang sorting order, a map such as `#{1.0 => true}` is going to compare equal to `#{1 => true}`, but you won't be able to match them one against the other.

Be careful, because although the content of this section is written as based on EEP-43, the actual implementation might be lagging behind!

Stubby Legs for Early Releases



The maps implementation that came with Erlang 17.x and 18.0 is complete, but only within the confines of the maps module. The major differences come from the syntax. Only a minimal subset of it is available:

- literal map declarations (`#{}, #{key1 => val1, "key2" => val2}`)
- matching on known keys (`#{a := X} = SomeMap`)
- Updating and adding elements with a known key in an existing map (`Map#{a := update, b => new}`)
- Using variables as keys both when assigning (`X = 3, #{X => X-1}`) and when matching (`#{X := 2} = #{3 => 2}`), starting in Erlang 18.0

The rest, including accessing single values (`Map#{key}`), whether in matches or declarations, is still not there. Same fate for map comprehensions and Dialyzer support. In fact, the syntax may change and differ from the EEP in the end.

Maps are also still slower than most Erlang developers and the OTP team want them to be. Still, progress is ongoing and it should not take long – especially compared to how long it's been before maps were added to the language in the first place – before they get much better.

This early release is still good enough to get familiar with maps, and more importantly, get to know when and where to use them.

Mexican Standoff

Everyone had their opinion on wanting native dictionaries or better records replacement as a priority. When maps were announced, many Erlang developers kind of just assumed that maps, when they came to the language, would solve the problem they wanted solved.

As such, there is some confusion on how maps should be used in Erlang.

Maps Vs. Records Vs. Dicts

To get it out of the way directly, maps are a replacement for dicts, not records. This can be confusing. At the beginning of this chapter, I identified common complaints, and it turns out that many of the complaints about records would be fixed by maps:

Issue Solved By Maps	Issue in Records	Issue in Dicts
Record Name is cumbersome	✓	
No good native k/v store		✓
Faster k/v store		18.x and above
Converting easier with native type	✓	✓
More powerful pattern matching		✓
Upgrades with Records	maybe	

Issue Solved By Maps	Issue in Records	Issue in Dicts
Usable across modules without includes	✓	



The score is pretty close. The point of maps being faster isn't necessarily the case yet, but optimizations should bring them to a better level. The OTP team is respecting the old slogan: first make it work, then make it beautiful, and only if you need to, make it fast. They're getting semantics and correctness out of the way first.

For upgrades with records being marked as 'maybe', this has to do with the `code_change` functionality. A lot of users are annoyed by having to convert records openly when they change versions similar to what we did with `pq_player.erl` and its upgrade code. Maps would instead allow us to just add fields as required to the entry and keep going. The counter-argument to that is that a botched up upgrade would crash early with records (a good thing!) whereas a botched up upgrade with maps may just linger on with the equivalent of corrupted data, and not show up until it's too late.

So why is it we should use maps as dicts and not as records? For this, a second table is necessary. This one is about *semantics*, and which data structure or data type a feature applies to:

Operations	Records	Maps	Dict
Immutable	✓	✓	✓
Keys of any type		✓	✓
Usable with maps/folds		✓	✓
Content opaque to other modules	✓		
Has a module to use it		✓	✓
Supports pattern matching	✓		✓
All keys known at compile-time	✓		
Can merge with other instance		✓	✓
Testing for presence of a key		✓	✓
Extract value by key	✓	✓	✓
Per-key Dialyzer type-checking	✓	*	
Conversion from/to lists		✓	✓
Per-element default values	✓		
Standalone data type at runtime		✓	
Fast direct-index access	✓		

* *The EEP recommends making this possible for keys known at compile-time, but has no ETA on when or if this will happen.*

This chart makes it rather obvious that despite the similarity in syntax between maps and records, dicts and maps are much closer together *semantically* than maps are to records.

Therefore, using maps to replace records would be similar to trying to replace 'structs' in a language like C with a hash map.

It's not impossible, but that doesn't mean it's a good idea given they often have different purposes.

Keys being known at compile time brings advantages with fast access to specific values (faster than what is possible dynamically), additional safety (crash early rather than corrupting state), and easier type checking. These make records absolutely appropriate for a process' internal state, despite the occasional burden of writing a more verbose `code_change` function.

On the other hand, where Erlang users would use records to represent complex nested key/value data structures (oddly similar to objects in object-oriented languages) that would frequently cross module boundaries, maps will help a lot. Records were the wrong tool for that job.

To put it briefly, places where records really felt out of place and cumbersome *could* be replaced by maps, but most record uses *should not* be replaced that way.

Don't Drink too Much Kool-Aid:

It is often tempting to bring complex nested data structures to the party and use them as one big transparent object to pass in and out of functions or processes, and to then just use pattern matching to extract what you need.

Remember, however, that there is a cost to doing these things in message passing: Erlang data is copied across processes and working that way might be expensive.

Similarly, passing big transparent objects in and out of functions should be done with care. Building a sensible interface (or API) is already difficult; if you marry yourself to your internal data representation, enormous amounts of flexibility for the implementation can be lost.

It is better to think about your interfaces and message-based protocols with great care, and to limit the amount of information and responsibility that gets shared around. Maps are a replacement for dicts, not for proper design.

There are also performance impacts to be expected. Richard O'Keefe mentions it in his proposal:

You can't make dict good for the record-like uses that frames are meant for without making it bad for its existing uses.

And the EEP from the OTP team also mentions something similar:

When comparing maps with records the drawbacks are easily remedied by maps, however the positive effects [*sic*] is not as easy to replicate in a built-in data-type where values are determined at runtime instead of at compile time.

- Being faster than direct-indexing array, where indices and possibly the resulting value are determined at compile time, is hard. In fact it is impossible.
- Memory model for maps where the efficiency is near that of records could be achieved by essentially using two tuples, one for keys and one for values as demonstrated in frames. This would impact performance of updates on

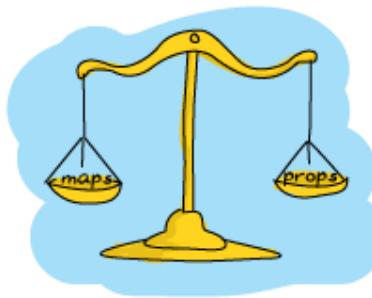
maps with a large number of entries and thus constrain the capability of a dictionary approach.

For the core of your process loop, when you know all keys that should exist, a record would be a smart choice, performance-wise.

Maps Vs. Proplists

One thing that maps may beat are proplists. A proplist is a rather simple data structure that is extremely well-suited for options passed to a module.

The `inet:setopts/2` call can take a list of options of the form `[{active, true}, binary]`, and the file handling functions can take argument lists such as `[read, write, append, {encoding, utf8}]`, for example. Both of these option lists can be read using the `proplists` module, and terms such as `write` will be expanded as if they were written as `{write, true}`.



Maps, with their single-key access (whenever implemented) will represent a similar way to define properties. For example, `[{active, true}]` can be expressed with maps as `#{active => true}`. This is equally cumbersome, but it will make reading the option much simpler, given you won't need to make a module call (thanks to `Opts#{active}`).

It's somewhat to be expected that option lists that are *mostly* pairs will be replaced by maps. On the other hand, literal options such as `read`, `write`, or `append` will remain much nicer with prolists, from the user's perspective.

Given that right now most functions that require options use prolists, keeping things going that way forward may be interesting for consistency's sake. On the other hand, when the options are mostly pairs, using the prolists module can get cumbersome fast. Ultimately, the author of a library will have to make a decision between what could be internal clarity of implementation or general ecosystem consistency. Alternatively, the author could decide to support both ways of doing things at once to provide a graceful path towards deprecation of prolists, if that's how they feel about it.

On the other hand, functions that used to pass prolists as a return value from functions should probably switch to maps. There is little legacy there, and usability should be much better for users that way.

Note: maps can use a clever trick to set many default values at once easily. Whereas prolists would require the use of `prolists:get_value(Key, List, Default)`, maps can use their `merge/2` function.

According to the specification, `merge/2` will fuse two maps together, and if two keys are the same, the second map's value will prevail. This allows to call `maps:merge(Default, YourMap)` and get the desired values. For example, `maps:merge(#{key => default, other => ok}, #{other => 42})` will result in the map `#{key => default, other => 42}`.

This makes for an extremely convenient way to attribute default values manually, and then you can just use the resulting map without worrying about missing keys.

How This Book Would Be Revised For Maps

I wanted to add this section because I do not necessarily have the time to update the book in its entirety to retrofit maps back in at this point in time.

For most of the book, however, little would change. It would mostly be replacing calls to the `dict` module and to `gb_trees` (in cases where the smallest/largest values aren't frequently required) with inline maps syntax. Pretty much none of the records used within the book would be changed, for semantics' sake.

I may revisit a few modules once maps are stable and fully implemented, but in the mean time, many examples would be impractical to write that way given the partial maps implementation.

Postscript: Time Goes On

On Time for Time

Time is a very, very tricky thing. In the physical every day world, we at least have one certainty: time moves forward, and generally at a constant rate. If we start looking at fancypants physics (anything where relativity is involved, so not *that* fancypants), then time starts drifting and shifting around. A clock on a plane goes slower than a clock on the ground, and someone nearing a black hole ages at a different speed from someone orbiting the moon.



Unfortunately for programmers and computer people, there is no need for nifty phenomena like that to be involved for time to go weird; clocks on computers are just not that great. They spring forwards, spring backwards, stall or accelerate, get leap seconds, get readjusted, and so on. On distributed systems, different processors run at different speeds, and

protocols such as NTP will play around with time corrections but may crash at any time.

There is therefore no need to leave the room for computer time to dilate and ruin your understanding of the world. Even on a single computer, it is possible for time to move in frustrating ways. It's just not very reliable.

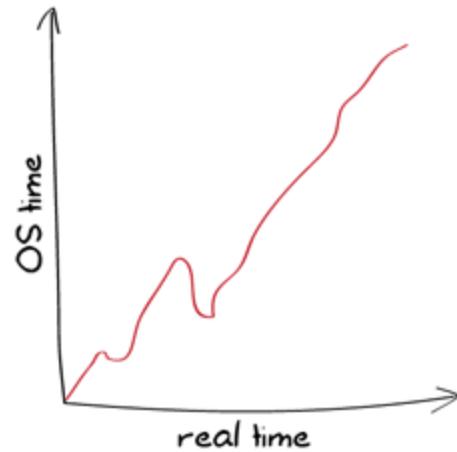
In the context of Erlang, we care a lot about time. We want low latencies, and we can specify timeouts and delays in milliseconds on almost every operation out there: sockets, message receiving, event scheduling, and so on. We also want fault tolerance and to be able to write reliable systems. The question is how can we make something solid out of such unreliable things? Erlang takes a somewhat unique approach, and since release 18, it has seen some very interesting evolution.

How Things Were

Before release 18, Erlang time works in one of two major ways:

1. The operating system's clock, represented as a tuple of the form {MegaSeconds, Seconds, MicroSeconds} (`os:timestamp()`)
2. The virtual machine's clock, represented as a tuple of the form {MegaSeconds, Seconds, MicroSeconds} (`erlang:now()`, auto-imported as `now()`)

The operating system clock can follow any pattern whatsoever:



It can move however the OS feels like moving it.

The VM's clock can only move forward, and can never return the same value twice. This is a property named being *strictly monotonic*:



In order for now() to respect such properties, it requires coordinated access by all Erlang processes. Whenever it is called twice in a row at close intervals or while time has gone backwards, the VM will increment microseconds to make sure the same value isn't returned twice. This coordination mechanism (acquiring locks and whatnot) can act as a bottleneck in a busy system.

Note: monotonicity comes in two main flavors: strict and non-strict.

Strict monotonic counters or clocks are guaranteed to return always increasing values (or always decreasing values). The sequence 1, 2, 3, 4, 5 is strictly monotonic.

Regular (non-strict) monotonic counters otherwise only require to return non-decreasing values (or non-increasing values). The sequence 1, 2, 2, 2, 3, 4 is monotonic, but not strictly monotonic.

Now, having time that never goes back is a useful property, but there are many cases when that is not enough. One of them is a common one encountered by people programming Erlang on their home laptop. You're sitting at the computer, running Erlang tasks at frequent intervals. This works well and has never failed you. But one day, you hear the chime of an ice cream truck outdoors, and put your computer to sleep before running outdoors to grab something to eat. After 15 minutes, you come back, wake

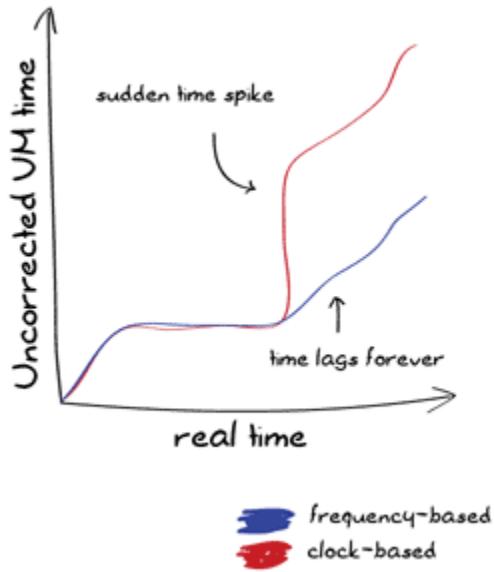
your laptop up, and everything starts exploding in your program. What happened?

The answer depends on how time is accounted for. If it's counted in cycles ("I have seen N instructions fly by on the CPU, that's 12 seconds!"), you could be fine. If it's counted by looking at the clock on the wall and going "gee golly, it's 6:15 now and it was 4:20 last time! that's 1h55 that went past!", then going to sleep would hurt a lot for tasks that are expected to run every few seconds or so.

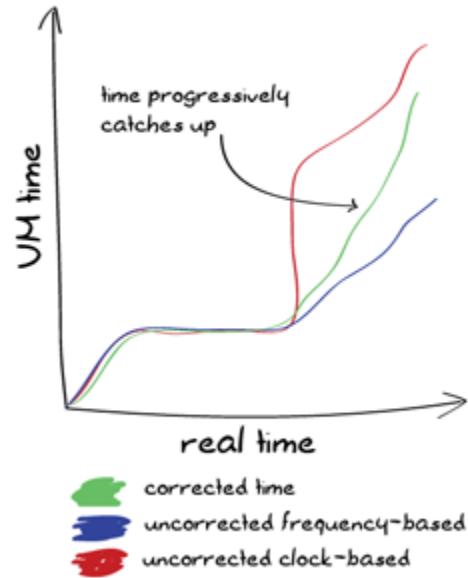
On the other hand, if you use cycles and keep them stable, you'll never really see the clock from the program synchronize up with the underlying operating systems. This means that either we can get an accurate `now()` value, or accurate intervals, but not both at once.

For this reason, the Erlang VM introduces *time correction*. Time correction makes it so the VM, for timers having to do with `now()`, after bits in `receive`, `erlang:start_timer/3` and `erlang:send_after/3` along with the `timer` module, will dampen sudden changes by adjusting the clock frequency to go slightly faster or slower.

So instead of seeing either of these curves:



We would see:



Time correction in versions prior to 18.0, if undesired, can be turned off by passing the `+c` argument to the Erlang VM.

How Things Are (18.0+)

The model seen for versions prior to 18.0 was fairly good, but it ended up being annoying in a few specific ways:

- Time correction was a compromise between skewed clocks and inaccurate clock frequencies. We would trade some accelerated or slowed frequency in order to get closer to the proper OS time. To avoid breaking events, the clock can only be corrected very slowly, so we could have both inaccurate clocks *and* inaccurate intervals for very long periods of time
- People used `now()` when they wanted monotonic *and* strictly monotonic time (useful to order events)
- people used `now()` when they wanted unique values (for the lifetime of a given node)
- Having the time as {MegaSecs, Secs, MicroSecs} is annoying and a remnant of times when bigger integers were impractical for the VM to represent and converting to proper time units is a pain. There is no good reason to use this format when Erlang integers can be of any size.
- Backwards leap in time would stall the Erlang clock (it would only progress microseconds at a time, between each call)

In general, the problem is that there were two tools (`os:timestamp()` and `now()`) to fill all of the following tasks:

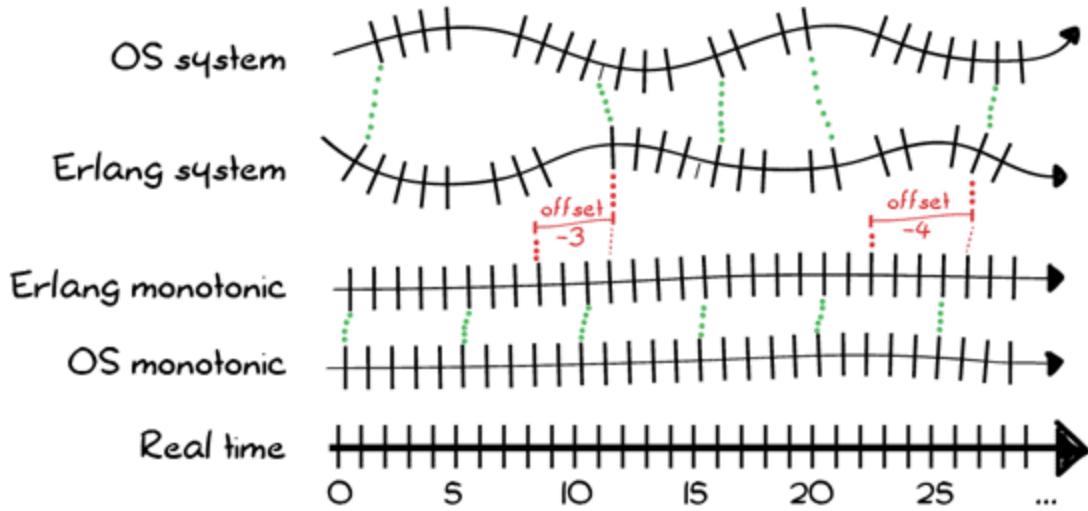
- Find the system's time
- Measure time elapsed between two events

- Determine the order of events (by tagging each event with `now()`)
- Create unique values

All of these are made clearer by exploding the time in Erlang into multiple components, starting in 18.0:

- OS system time, also known as the POSIX time.
- OS Monotonic time; some operating systems provide it, some don't. It tends to be fairly stable when available, and avoids leaps in time.
- Erlang system time. It's the VM's take on POSIX time. The VM will try to align it with POSIX, but it may move around a bit depending on the chosen strategy (the strategies are described in [Time Warp](#)).
- Erlang monotonic time. This is Erlang's view of the OS monotonic time if available, or the VM's own monotonic version of the system time if not available. This is the adjusted clock that is used for events, timers, and so on. Its stability makes it ideal to count a time interval. Do note that this time is *monotonic*, but not *strictly monotonic*, meaning that the clock can't go backwards, but it can return the same value many times!
- Time offset; because the Erlang Monotonic time is a stable source of authority, the Erlang system time will be calculated by having a given offset relative to the Erlang monotonic time. The reason for this is that it will allow Erlang to adjust the system time without modifying the monotonic time frequency.

Or more visually:



If the offset is a constant 0, then the VM's monotonic and system times will be the same. If the offset is modified positively or negatively, the Erlang system time may be made to match the OS system time while the Erlang monotonic time is left independent. In practice, it is possible for the monotonic clock to be some large negative number, and the system clock to be modified by the offset to represent the positive POSIX timestamp.

With all these new components, another use case remains: unique values that *always* increment. The high cost of the now() function was due to this necessity that it never returns the same number twice. As mentioned earlier, the Erlang monotonic time is not *strictly* monotonic: it will possibly return the same number twice if it's called at the same time on two different cores, for example. By comparison, now()

wouldn't. To compensate for this, a strictly monotonic number generator was added to the VM, so that time and unique integers could be handled separately.

The new components of the VM are exposed to the user with the following functions:

- `erlang:monotonic_time()` and `erlang:monotonic_time(Unit)` for the Erlang monotonic time. It may return very low negative numbers, but they'll never get more negative.
- `erlang:system_time()` and `erlang:system_time(Unit)`, for the Erlang system time (after the offset has been applied)
- `erlang:timestamp()` returns the Erlang system time under the {MegaSecs, Secs, MicroSecs} format for backwards compatibility
- `erlang:time_offset()` and `erlang:time_offset(Unit)` to figure out the difference between the Erlang monotonic and Erlang system clocks
- `erlang:unique_integer()` and `erlang:unique_integer(Options)`, which returns unique values. The *Options* list can contain either or both of `positive` (to force numbers greater than 0) and `monotonic` (so they always grow larger). *Options* defaults to `[]`, which means that while the integers are unique, they might be positive or negative, and greater or smaller than previous ones given.
- `erlang:system_info(os_system_time_source)`, which gives access to the tolerance, intervals, and values of the OS system time.

- `erlang:system_info(os_monotonic_time_source)`: if the OS has a monotonic clock, its tolerance, intervals, and values can be fetched there.

The *Unit* option in all the functions above can be either seconds, milli_seconds, micro_seconds, nano_seconds, or native. By default, the type of timestamp returned is in the native format. The unit is determined at run time, and a function to convert between time units may be used to convert between them:

```
1> erlang:convert_time_unit(1, seconds, native).  
1000000000
```

Meaning that my linux VPS has a unit in nanoseconds. The actual resolution may be lower than that (it's possible that only milliseconds are accurate), but nonetheless, it natively works in nanoseconds.

The last tool in the arsenal is a new type of monitor, usable to detect when the time offset jumps. It can be called as `erlang:monitor(time_offset, clock_service)`. It returns a reference and when time drifts, the message received will be `{'CHANGE', MonitorRef, time_offset, clock_service, NewTimeOffset}`.

So how does time get adjusted? Get ready for time warp!

Time Warp



The old style Erlang stuff would just make clocks drift faster and slower until they matched whatever the OS gave. This was okay to keep some semblance of real time when clocks jumped around, but also meant that over time, events and timeouts would occur faster and slower by a small percentage across multiple nodes. You also had a single switch for the VM, `+c`, which disabled time correction altogether.

Erlang 18.0 introduces a distinction between how things are done and makes it a lot more powerful and complex. Whereas versions prior to 18.0 only had time drift, meaning the clocks would accelerate or slow down, 18.0 introduced both time correction and a thing called *time warp*.

Basically, *time warp*, configured with `+c`, is about choosing how the *offset* (and therefore the *Erlang system time*) jumps around to stay aligned with the OS. Time warp is a time jump. Then there's *time correction*, configured with `+c`, which is how the *Erlang monotonic time* behaves when the OS monotonic clock jumps.

There's only two strategies for time correction, but there's three for time warp. The problem is that the time warp strategy chosen impacts the time correction impact, and therefore we end up with a stunning 6 possible behaviours. To make sense out of this the following table might help:

		Works exactly as it did before 18.0. Time does not warp (does not jump), but clock frequency is adjusted to compensate. This is the default, for backwards compatibility.
+C	no_time_warp	If the OS system time jumps backwards, +c the Erlang Monotonic clock stalls until false the OS system time jumps back forward, which can take a while.
+C		The Erlang system time is adjusted backwards and forwards via the offset +c to match the OS system time. The true monotonic clock can remain as stable and accurate as possible.
+C	multi_time_warp	The Erlang system time is adjusted backwards and forwards via the offset, +c but because there's no time correction, false the monotonic clock may pause briefly (without freezing long).
+C	single_time_warp	This is a special hybrid mode to be used on embedded hardware when you know Erlang

boots before the OS clock is synchronized. It works in two phases:

1. a. (with `+c true`) When the system boots, the monotonic clock is kept as stable as possible, but no system time adjustments are made
b. (with `+c false`) Same as `no_time_warp`
2. The user calls `erlang:system_flag(time_offset, finalize)`, the Erlang system time warps once to match the OS system time, and then the clocks become equivalent to those under `no_time_warp`.

Whew. In short, the best course of action is to make sure your code can deal with time warping, and go into multi time warp mode. If your code isn't safe, stick to no time warp.

How to Survive Time Warps

- To find system time: `erlang:system_time/0-1`
- To measure time differences: call `erlang:monotonic_time/0-1` twice and subtract them
- To define an absolute order between events on a node:
`erlang:unique_integer([monotonic])`
- Measure time *and* make sure an absolute order is defined: `{erlang:monotonic_time(), erlang:unique_integer([monotonic])}`

- Create a unique name: `erlang:unique_integer([positive])`.
Couple it with a node name if you want the value to be unique in a cluster, or try using UUIDv1s.

By following these concepts, your code should be fine to use in multi time warp mode with time correction enabled, and benefit from its better accuracy and lower overhead.

With all this information in hand, you should now be able to drift and warp through time!