Thinking in Erlang

A GUIDE TO FUNCTIONAL PROGRAMMING IN ERLANG FOR THE EXPERIENCED PROCEDURAL DEVELOPER

Robert Baruch
autophile@zoominternet.net
Version 0.9.1
February 5, 2007

Copyright

Contents

1 Introduction

1.1 What this document is not

If you expected to learn programming from this document, you will be disappointed. You are expected to know either C++, C#, or Java fairly well. This document is also not a reference for Erlang. There's plenty of Erlang documentation, so there's no sense in padding this one with duplicated information.

Furthermore, we're not going to justify the use of Erlang over any other particular programming language, functional or otherwise. It is assumed that you're reading this document because you want to do something in Erlang, which implies you've already made the decision to use it, or at least try it out.

Finally, you won't find an Erlang installation manual here. Again, there are plenty of resources on the web for you to install Erlang on your particular environment.

1.2 What this document is

Erlang is a *functional programming language*. This is very different from C++, C#, and Java, which are *procedural programming languages*. Procedural languages stress organization of data, and sequences of instructions which operate on some global state. Functional languages, on the other hand, treat programs as evaluations of *functions* without a global state.

Because there is no global state, functional programming languages may seem bizarre to the expert procedural software architect. We'll try to tie in concepts from Java to Erlang to make understanding easier. The author is a Java veteran, has long abandoned C++, and has never seen the need to move to C#, so most of the analogies will be drawn from Java.

1.3 Porting code to Erlang

Porting code from a procedural language to a functional language is not easy. You have to understand what the code is trying to accomplish. Because the paradigms are so different, you will be designing the new code from scratch. This is why it is important to know how to think in Erlang.

1.4 Hello, World

You knew it was coming, so let's get it out of the way.

```
1 -module(hello).
2 -export([hello/0]).
3
4 hello() ->
5     io:fwrite("Hello, World!~n", []).
```

Figure 1: Hello, World!

Very strange-looking, but let's tie this to things you already know. Line 1, the module declaration, is like a class declaration. It says that this file, or *module*, is named hello. Note that unlike C++ or Java, module names must begin with a lower-case letter. This is because Erlang words beginning with an upper-case letter are used for variables only.

Line 2 shows which *functions* are allowed to be called from outside the module. A function, by the way, is Erlang-speak for method. Thus, with export, we have the equivalent of public and private methods. Since there is no inheritance in Erlang, there is no equivalent to protected methods.

The brackets in the export statement indicate a *list*. Erlang lists are like arrays, but they may grow and shrink, and have elements inserted and deleted. Until we get to more complicated examples later, just accept for now that this list has one element.

The hello/0 shows that we are going to export the function named hello, in its 0-arity incarnation. *Arity* is simply the number of arguments a function takes. You can have two functions of the same name with different arities, but two functions of the same name with the same arity are actually the same function. We'll see that later as well.

Line 4 is a function declaration. The arrow indicates that we are going to begin the body of the function, which ends in a period.

Line 5 is the body. We are calling the fwrite function in the io module. io:fwrite is like printf in C or PrintStream.printf in Java (1.5 or above). The first argument is the format string, while the second argument is a list of things the format string will use. The empty brackets here show that the list is empty.

The tilde character (~) is equivalent to the % character in C's printf or Java's PrintStream.printf. ~n is the format specifier to output an environment-specific newline –

although in practice, only a linefeed (ASCII 0x0A) is emitted, so Windows environments (and, I guess, IBM OS/390 systems which use NEL instead of LF or CR) are not well-supported.

1.5 Compiling and running hello.erl

Let's tackle compiling and running hello.erl in the Erlang interactive shell:

```
ekmac:~ ek$ erl
Erlang (BEAM) emulator version 5.5.2 [source] [async-threads:0] [hipe] [kernel-poll:false]

Eshell V5.5.2  (abort with ^G)
1> c("hello.erl").
{ok,hello}
2> hello:hello().
Hello, World!
ok
3> init:stop().
ok
4> ekmac:~ ek$
```

Figure 2: Running hello.erl

The first command we issue compiles hello.erl from the current directory. You can always put in an absolute or relative path here. Note that this command takes the form of a function call, and ends with a period.

Next, we call the hello/0 function in the module hello. This is very similar to calling a static method on a class. Again, we end the call with a period. The "ok" at the end is just the return value of hello/0 – functions return the value of the last statement, which in this case was io:fwrite/2, which returns the value ok.

Finally, to shut down and exit the shell, we call the function init:stop/0.

2 Scope of variables

2.1 No global state

As stated briefly earlier, Erlang has no global state. This means no global variables. And while we also said that modules were analogous to classes, although classes can have attributes, modules cannot have variables.

The scope of an Erlang variable is limited to the function it is declared in. Furthermore, once defined, it cannot be assigned to again: that would imply that the function has a state, which it does not.

Finally, you don't declare Erlang variables as being of a particular type. The type is determined at runtime, since there isn't any way to confuse one type with another, and there is no way, within the same scope, to retype a variable. Here's an example.

```
Eshell V5.5.2  (abort with ^G)
1> X = 1.
1
2> X.
1
3> X = 2.

=ERROR REPORT==== 17-Jan-2007::19:58:46 ===
Error in process <0.30.0> with exit value: {{badmatch,2},[{erl_eval,expr,3}]}

** exited: {{badmatch,2},[{erl_eval,expr,3}]} **
4> F = 1.0.
1.00000
5> S = "abc".
"abc"
6> L = [1, 2, 3].
[1,2,3]
7> L2 = [$a, $b, $c].
"abc"
```

Figure 3: Various variables

When a variable is assigned a value, it is said to be *bound*. Once a variable is bound, it cannot be bound again. Paraphrasing a US President: "Bind me once... good on... good on you. Bind me... you can't get bound again!"

In line 2, we are asking to read back the value of X. Line 3 gives an error because you cannot rebind variables in the same scope. The error returned is "badmatch", which will we will explain

later.

Line 4 shows a float, line 5 shows a string, line 6 shows a list, and line 7 shows a list of characters ($x is the Erlang equivalent of C and Java 'x', so that $\n is a newline character and $$ is a $ character).

Note that L2 displays as a string. This is because there isn't really a string type in Erlang. Strings are implemented as lists of characters – in fact, there is no character type, either, so strings are really just lists of integers. So although you could have a list of 16-bit integers and call it a Unicode string, Erlang's support for Unicode is nonexistent. Working with Unicode in Erlang is beyond the scope of this document. Suffice it to say that while representing a Unicode string is trivial, comparing two Unicode strings using proper Unicode normalization, or even determining whether a Unicode character is whitespace, is not implemented.

2.2 Dealing with no global state

By this time, your brain has probably siezed up. Your typical C or Java program must use `i++` at least once[1]. And yet you can't do that in Erlang, because it implies that you are changing the value of a variable.

This is where thinking in Erlang (or thinking functionally) comes in. Leaving aside `i++`, let's look at `S /= 1000`. Perhaps S is in milliseconds and you want to convert it to seconds. Because we're using Erlang, this would have to be implemented as something like `S2 = S / 1000`.

At this point there's a riot going on. "You just doubled your memory requirement!" someone shouts. "That's wasteful!" another one yells. People start turning over police cars to chants of "Erlang sucks!"

Well, why would we keep S around after the S2 assignment? Do we need both S and S2? If so, we have no choice but to keep them both around. If we don't need S, then after the assignment it can be thrown away. Or, if we only need S2 once, we could even dispense with S2 and use a temporary where we need it. Furthermore, once the function where S was defined is ended, we no longer need S.

The point is that once you get rid of reassignments, the compiler's optimization job becomes much easier.

[1] You usually see this in loops. See the chapter on Loops for this issue

Of course, if you try this in the Erlang shell, the shell has no choice but to keep S around because it doesn't know what you are going to do next.

3 Matching

3.1 Basic matching

Suppose you had a tuple of three elements. A tuple is like a list, except you generally don't grow or shrink it, and it does not have a tail like a list does (more about that later). Tuples are represented in Erlang by curly braces, thus `{ 1, 2, "burger" }` is a tuple consisting of three elements. The first two are integers, and the third is a list.

Now what if this tuple were passed to your function called middle/1, which takes this as a variable, thus: middle(Arg). You want to return the second element.

You could use erlang:element/2, which returns the Nth element of a give tuple. This will work for tuples of any size. But if you know that your tuple always consists of three elements, there is an easier way:

```
1 middle(Arg) ->
2     { _First, Second, _Third} = Arg,
3     Second.
```

Figure 4: Matching a tuple

The assignment in line 2 is a *match*. First of all, if Arg is not a tuple of three elements, the match will fail with a badmatch error, and the function will fail. But that's OK, because we will only pass middle/1 a tuple of three arguments (but see below if you don't!).

Secondly, each element in the tuple will be matched up against the left hand side, and bound. The variables that begin with an underscore are also bound, but is an indication that the value is never actually used in the scope. You could also just as easily have written `{ _, Second, _ }`, which makes use of the special variable _ which is never bound, but that may not be as clear, depending on the context.

Thus, given `{ 1, 2, "burger" }`, the match will assign 2 to Second, and ignore the other elements. The function then returns Second.

As mentioned above, if middle/1 is passed anything other than a tuple or a tuple without three elements, then middle/1 will fail with a badmatch error:

```
2> test:middle( { 1, 2, "burger" } ).
2
3> test:middle( "burger" ).

=ERROR REPORT==== 18-Jan-2007::19:08:39 ===
Error in process <0.30.0> with exit value: {{badmatch,"burger"},
                [{test,middle,1},{shell,exprs,6},{shell,eval_loop,3}]}

** exited: {{badmatch,"burger"},
            [{test,middle,1},{shell,exprs,6},{shell,eval_loop,3}]} **
```

Figure 5: A bad match

The badmatch information tells us that we tried to match "burger" against something that failed at test:middle/1 (which was called from shell:exprs/6, which was called from shell:eval_loop/3)[2].

While you cannot match just the second element in a tuple of any arbitrary size without calling erlang:element/2, you can match arbitrary lists. Here is an example showing list matching:

```
1 middle( [ _First, Second |_Tail] ) ->
2     Second.
```

Figure 6: Matching a list

This syntax in line 1 shows that we must have two elements, followed by the rest of the list. The rest of the list is a list, so matching [Head | Tail] to [1, 2, 3, 4, 5] would set Head to 1 and Tail to [2, 3, 4, 5]. If the list has no tail, then the tail gets set to the empty list, [ ]. This means that matching [Head | Tail] to [1] would set Head to 1 and Tail to [ ]. Matching [Head | Tail] to [ ] would fail, because Head needs an element, and there are no elements in the list.

So you can see that line 1 requires the list to have at least two elements. If the list has less than two elements, _First and Second could not match, and you would end up with a badmatch error.

[2] See the chapter on Dealing with errors for more information.

3.2 Function argument matching

When trying to determine which function to run, Erlang matches not only the name of the function, but also the arguments given by the caller to the function. The following example shows the tuple-matching example in a more compact way using argument matching:

```
1 middle( { _First, Second, _Third} ) ->
2     Second.
```

Figure 7: Matching in function arguments

We could even combine the two previous examples – extracting the second element in a tuple, and extracting the second element in a list – by defining multiple alternatives for the same function:

```
1 middle( { _First, Second, _Third} ) ->
2     Second;
3 middle( [ _First, Second, |_Tail ] ) ->
4     Second.
```

Figure 8: Matching and function alternatives

On lines 1 and 3 we have defined two middle/1 functions. For each function call, Erlang goes through each alternative for the function, attempting to match the arguments to the alternative. Here we see we have two alternatives. The first one only matches if the passed argument is a 3-tuple. The second matches only for a list of at least two elements.

It is important to order the alternatives properly. Erlang will stop at the first match it finds, so make sure you order the alternatives from most restrictive to least restrictive.

Notice also that the first alternative does not end in a period, but in a semicolon. That tells the compiler that another alternative for the function is coming. If we had used a period on line 2, the compiler would think that line 3 is a redefinition of middle/1, which would be an error.

3.3 If and case

As in C and Java, there are if statements and switch statements – these are called *if* and *case* in Erlang. If and case have return values, which are the result of the body that is executed when one of the choices is true (for if) or matches (for case). Here is an example:

```
1 is_even(X) ->
2     if
3         X rem 2 == 0 ->
4             true;
5         true ->
6             false
7     end.
```

Figure 9: Example of *if*

You can have as many clauses in an if-statement as you like. Erlang will check each one, and the first one that evaluates as true has its body evaluated and returned. Because the if statement is a value-returning statement, you could assign the result to a variable and use it further down in the function, i.e. `Ret = if...end`.

The case statement matches against patterns:

```
1 many(X) ->
2     case X of
3         [ ] ->
4             none;
5         [ _One ] ->
6             one;
7         [ _One, _Two ] ->
8             two;
9         [ _One, _Two, _Three |_Tail ] ->
10            many
11    end.
```

Figure 10: Example of *case*

Here we are matching X against patterns. The first pattern to match has its body evaluated and returned. The return values here are not strings, because they are not surrounded by double quotes. Nor are they variables, because they do not begin with an upper case letter. They are *atoms*, which are the equivalent of enumerated values in a global enumeration space. Atoms can be created anywhere, passed around, matched, and so on. Converting an atom to its name is done using erlang:atom_to_list/1.

In many cases, atoms are more efficient than strings, and more descriptive than integers, so consider their use where you would think of using a string or an integer as an enumerated value.

Atoms can also be written using single quotes: ‘none’ instead of none. This allows you to have an atom beginning with an upper case letter, or even an atom that is a symbol, such as ‘:’, which is an atom whose name is “:”. But we digress.

If you are wondering why line 9 is not a match against [_One, _Two | _Tail], review the list

matching rules for list tails at the end of the previous section.

The syntax of if and case is important to understand. Bodies which are followed by another alternative must end in a semicolon, and the last body does not end in anything except an end token. If the if or case statement is followed sequentially by another statement, end must be followed by a comma. If the if or case statement is the last statement in a function, it must end in a period (or a semicolon if an alternative to the function follows).

So generalizing, a semicolon indicates that an alternative is coming, and a period (or end for if and case) indicates no more alternatives. Remember this, and you will be thinking in Erlang.

3.4 Guards

Functions and cases can use additional checks to see if they should be executed. These are called *guard sequences*, which are boolean expressions with the additional limitation that they may not call functions, except for a very limited set of functions detailed in the Erlang manual. For example, we could rework the is_even function to use guards:

```
1 is_even(X) when X rem 2 == 0 ->
2     true;
3 is_even(_X) ->
4     false.
```

Figure 11: Example of a guard sequence

Guard sequences are even more useful when used in conjunction with pattern matching. A pattern can match when a list has at least one element, but it cannot match if that element is an even integer or not. A guard used with a pattern can do that.

Guard sequences can get very complex. The sequence used above consists of a single guard expression, but in general, a *guard sequence* is a series of *guards*, separated by semicolons, any one of which must be true. Each *guard* is a series of *guard expressions*, separated by commas, all of which must be true. The Erlang manual goes into more detail about the syntax of guard sequences.

4 Loops

4.1 Recursion

Okay, without `i++`, how does Erlang implement loops? The short answer is, through recursion.

We've just touched off another riot.

Just as we argued in the previous section that lack of reassignment doesn't hurt, recursion also doesn't hurt as long as the recursion is tail recursion.

Tail recursion means that if a function calls itself, then that is the last thing it does. If calling itself (or any other function) is the last thing a function does, as soon as it sets up the arguments to that last function call, it may eliminate any variables that had been defined within the function's scope – which includes the function's arguments as well. Thus, with tail recursion, a function uses no additional memory.

When you want to implement a loop, you must include the loop variable in the function that is performing the loop. And that generally means that it is wise to define a function which only performs the loop (as opposed to a function which does something, then a loop, then something else).

Here's an example of printing the integers from 1 to 10 using tail recursion. We'll use argument matching, but also a construct you haven't seen before: *sequencing*.

```
1  -module(count).
2  -export([go/0]).
3
4  go() ->
5      go(1).
6
7  go(11) ->
8      io:fwrite("~n", []);
9  go(X) ->
10     io:fwrite("~.10B ", [X]),
11     go(X+1).
```

Figure 12: Counting example

This module defines two functions: go/0 and go/1. They are completely separate functions. They have no relation to each other. go/0 simply calls go/1 with an argument of 1, which is where we want the loop to start.

On line 10, we see that the io:fwrite statement has ended with a comma instead of a period. Statements can be executed sequentially by using a comma between the statements. And although it doesn't matter here, the return value of a block of sequenced statements is the return value of the last statement. This is the same as in C and Java.

Now would be a good time to familiarize yourself with the Erlang reference documentation to figure out what $\sim$.10B means. Hint, check the modules page, and find the io module. You are certainly welcome to create a more easily navigable API document.

Note that the recursive call to go/1 is the last thing that go/1 does. This means that go/1 properly implements tail recursion. During the execution of line 11, the argument X+1 is set up, the local variable X may now be discarded, and go/1 may be called again. We don't even need to implement this as a call. It may as well be a goto. Thus, no extra stack space is used for tail-recursive calls.

If, on the other hand, we had reversed lines 10 and 11 so that we called go(X+1) before printing out X (thus printing the numbers backwards), then go/1 would no longer be tail recursive, and we would have to store the arguments and the callee on the stack. So try to make your functions tail-recursive.

Here's the canonical non-tail-recursive example:

```
1 -module(badFactorial).
2 -export([factorial/1]).
3
4 factorial(0) -> 1.
5 factorial(N) ->
6     N * factorial(N-1).
```

Figure 13: Bad recursion

Although it may look like the call to factorial/1 is the last thing factorial/1 does, it is not. In fact, after factorial(N-1) returns, factorial(N) must multiply the result by N and then return. Since this is not tail-recursive, this function is wasteful of memory.

Functions that are recursive but not tail-recursive can sometimes be "cured" by the use of an additional accumulator argument which totals up a result. Here is the factorial example with an accumulator, demonstrating tail recursion:

```
 1 -module(goodFactorial).
 2 -export([factorial/1]).
 3
 4 factorial(N) ->
 5     factorial(N, 1).
 6
 7 factorial(0, Acc) ->
 8     Acc,
 9 factorial(N, Acc) ->
10     factorial(N-1, N*Acc).
```

Figure 14: Tail recursion properly implemented

There are three functions here. I like to call them the "setup" function, the "termination" function, and the "recursion" function. The setup function is exported, while the others are kept private.

We can see that factorial/2 is now tail-recursive. Once the arguments are computed, the only thing left to do is call factorial/2 with the arguments, and return whatever it returns. We can also see that factorial/2 carries an accumulator around, which totals up the result so far. We count down from N, multiplying the accumulator (which has been initialized in the setup with 1) by N, until we hit N=0 at which point there is nothing to do (the termination), so we just return what has been accumulated.

It may be difficult at first to understand the concept of accumulators as arguments, but after a few tries at your own functions, you will get the hang of thinking in Erlang.

What happens if N is negative? We go into an infinite loop. To prevent this, you would use a guard, which we will get to later.

Sometimes tail-recursion is not appropriate or worth it for some recursive problems. For example, traversing a left-right tree may be implemented recursively. But because at each node you must traverse the left node and then the right node, the traversal cannot be tail-recursive because traversing the left node is not the last call that the traversal makes.

One could always create a tail-recursive tree traversal, but the code and the thought that goes into it may be odd and difficult to follow. I highly recommend staying away from odd, tortuous code, unless your purpose is to create efficiency where it isn't needed, to create a maintenance nightmare, or to prove how clever you are – in which case you probably shouldn't be programming at all.

4.2 Less painful loops

When looping over a list, you can take advantage of three efficient list functions, which are map, foreach, and fold. These functions can sometimes obviate the need to create looping functions and accumulators.

The *map* function simply returns a list, each of whose elements is the result of applying a supplied function to the corresponding element of a supplied list. For example, if we had a list of 3-tuples (i.e. tuples of three elements), we could extract the second element of each tuple as follows:

```
1 -module(mapping).
2 -export([extract/1]).
3
4 extract(List) ->
5     lists:map(fun extractFromTuple/1, List).
6
7 extractFromTable( {_, Second, _} ) ->
8     Second.
```

Figure 15: Example of *map*

```
26> c("mapping.erl").
{ok,mapping}
27> mapping:extract( [ {1, 2, 3}, {4, 5, 6} ] ).
[2,5]
```

Figure 16: Using the example

If the function argument to the map function is small enough to be readable, we could insert the entire body of the function into the call to map. This would make it an anonymous function, which should sound familiar to Java programmers:

```
1 -module(mapping).
2 -export([extract/1]).
3
4 extract(List) ->
5     lists:map(fun ( { _, Second, _} ) -> Second end, List).
```

Figure 17: Example of anonymous functions

The *foreach* function is similar to the map function, except there is no value returned.

The *foldl* and *foldr* functions take a function and two additional arguments. The function argument must take two arguments, the first being an element, and the second being an accumulator, and must return the new value of the accumulator based on the value of the element. The two arguments to foldl and foldr are the initial value for the accumulator, and the list on which to perform the fold. The foldl function traverses the list from the beginning to the end, while the foldr function traverses the list in the opposite direction. foldl is preferred to foldr because foldl is tail-recursive.

Here is an example which counts the number of a's in a string.

```
1 -module(count).
2 -export([count/1]).
3
4 count(String) ->
5     lists:foldl(fun (Element, Acc) ->
6         case Element of
7             $a ->
8                 Acc + 1;
9             _->
10                Acc
11        end
12    end,
13    0,
14    String).
```

Figure 18: Example of *foldl*

```
42> count:count("aabdca").
3
```

Figure 19: Using the example

4.3 Summary of less painful loops

If you need to perform an action for each element in a list (returning no value), use *lists:foreach*.

If you need to compute a value for each element in a list, use *lists:map*.

If you need to accumulate a value for each element in a list, use *lists:foldl* or *lists:foldr*.

There are several other useful less painful loops provided in the lists module. It is a good idea to glance at the lists documentation to gain some awareness of what is available.

5 Processes

5.1 Doing two things at once

Just as we have threads in Java, we also have *processes* in Erlang. However, since there is no global state, Erlang threads are extremely light-weight. As the manual says, Erlang is designed for massive concurrency.

In Java, we can start a thread by creating a Thread subclass with the run method defined (or creating a class implementing the Runnable interface), then calling its start method. Erlang has no such requirement: any exported function can be used as an entry point, and the process ends when the function terminates.

To start a process, simply call *spawn*, giving it the name of the function to execute in a separate process, along with the arguments to pass to the function. The spawn function returns a process ID value which may be used in other functions. The arguments to the function to be called are matched against the arguments given.

```
1 -module(process1).
2 -export([main/0, thread/0]).
3
4 main() ->
5     Pid = spawn(process1, thread, []),
6     io:fwrite("Spawned new process ~w~n", [Pid]).
7
8 thread() ->
9     io:fwrite("This is a thread.~n", []).
10
```

Figure 20: Example of spawning a process

```
9> c("process1").
{ok,process1}
10> process1:main().
Spawned new process <0.65.0>
This is a thread.
ok
```

Figure 21: Result of spawning a process

5.2 Interprocess communication

Each process has a *mailbox*, which is a single queue of received *messages*. Messages may be anything: a string, an atom, an integer, a tuple, a list, and so on. The most useful things to send a process are atoms and tuples whose first element is an atom. Then the atom can be the name of the message, and any other elements are additional message data.

A process sends a message to another process using the *send* construct: `Pid ! Message`. Even if the Pid doesn't exist, the send will succeed.

A process can block to receive a message by using the *receive* construct. It works just like case: it takes any number of patterns which are matched against the first received message. The first pattern that matches the message will have its body executed. However, unlike case, if a message does not match, the message is replaced on the queue, and the process blocks once more, waiting for a message that does match.

This is important: unmatched messages are not discarded, but kept on the queue until such time as they can be matched by some other receive construct.

The receive construct may also take an optional timeout value in millseconds, after which the "after" clause of the receive construct is executed. In this sense, the after clause is just another pattern to match against, except this pattern must come at the end of the receive construct. See the manual for detailed syntax: the body before the after clause is *not* terminated by a semicolon.

```
1  -module(receive1).
2  -export([main/0, thread/0]).
3
4  main() ->
5      Pid = spawn(receive1, thread, []),
6      io:fwrite("Spawned new process ~w~n", [Pid]),
7      Pid !  hello.
8
9  thread() ->
10     io:fwrite("This is a thread.~n", []),
11     process_messages().
12
13 process_messages() ->
14     receive
15     hello ->
16         io:fwrite("Received hello~n"),
17         process_messages()
18     after 2000 ->
19         io:fwrite("Timeout.~n")
20     end.
21
```

Figure 22: Example of interprocess communication

```
7> receive1:main().
Spawned new process <0.53.0>
This is a thread.
Received hello
hello
Timeout.
```

Figure 23: Running the example

Notice how process_messages is tail-recursive.

In the output, we can see that we spawn a new process, the process starts, the spawned process receives the hello message, the original process terminates (the return value of send is the sent message, so the output is `hello`), and finally, after two seconds, the spawned process times out.

5.3 Process termination reasons

We saw that we can have a process terminate itself normally by simply ending the process's function. We can have a process terminate itself immediately by calling a termination function, such as *exit* or *erlang:error*. Each of these takes a reason code. A process can terminate itself normally by setting the reason code to `normal`.

When a process terminates itself for any reason other than normal, an *exit signal* to be sent to all *linked* processes. Two processes may be linked together (bidirectionally) when one process calls the `link` function with the other process's ID, and they may be unlinked by calling the `unlink` function. See the chapter on Dealing with errors for more detail about this topic.

A process can terminate another process by calling the exit function with a process ID and a reason code. However, in this case, an exit signal will not be sent when the process terminates.

```
-module(receive2).
-export([main/0, thread/0]).

main() ->
    Pid = spawn(receive2, thread, []),
    io:fwrite("Spawned new process ~w~n", [Pid]),
    Pid !  hello,
    exit(Pid, suckage).

thread() ->
    io:fwrite("This is a thread.~n", []),
    process_messages().

process_messages() ->
    receive
    hello ->
        io:fwrite("Received hello~n"),
        process_messages()
    after 2000 ->
        io:fwrite("Timeout.~n")
    end.

```

Figure 24: Example of killing another process

```
6> receive2:main().
Spawned new process <0.51.0>
This is a thread.
true
```

Figure 25: Running the example

5.4 Processes as objects

It is possible to consider a process as an object. If the data were stored in the arguments of the entry point, and the messages that the process received were instructions to operate on the data, then what you have is, in effect, an object. Here's a silly example of wrapping a data element inside an object, with constructor, destructor, set, and get methods:

```erlang
-module(object).
-export([main/0, new/1, get/1, set/2, delete/1, construct/1]).

% Testing

main() ->
    Object = new(1),
    io:fwrite("Get data:  ~w~n", [object:get(Object)]),
    set(Object, 2),
    io:fwrite("Get data:  ~w~n", [object:get(Object)]),
    delete(Object).

% Interface

new(Thing) ->
    spawn(object, construct, [Thing]).

get(Object) ->
    Object !  {get, self()},
    receive
    {return, Object, Thing} ->
        Thing
    end.

set(Object, Thing) ->
    Object !  {set, self(), Thing}.

delete(Object) ->
    exit(Object).

% Internals

construct(Thing) ->
    io:fwrite("Called constructor:  ~w~n", [Thing]),
    process_flag(trap_exit, true),
    process_messages(Thing).

process_messages(Thing) ->
    receive
    {get, Caller} ->
        io:fwrite("Called get~n"),
        Caller !  {return, self(), Thing},
        process_messages(Thing);
    {set, _Caller, NewThing} ->
        io:fwrite("Called set~n", []),
        process_messages(NewThing);
    {'EXIT', _Caller, _Reason} ->
        io:fwrite("Called destructor~n", []),
        true
    end.

```

Figure 26: A silly object

```
2> object:main().
Called constructor: 1
Called get
Get data: 1
Called set
Called get
Get data: 2
** exited: <0.37.0> **
```

Figure 27: Putting the object through its paces

Note the synchronous get function. Even if there are messages from other processes, they will remain in the queue until they can get processed. In practice, flushing unknown messages is sometimes a good idea, by having a catch-all match at the end of the receive clause, which throws the message away, and perhaps logs the bad message.

On line 8, we must explicitly use the module name to call the right function, because there is a *get* function polluting the global namespace. This function, like all global functions, is documented in the erlang module.

Note that on line 33, the argument is a single Thing, because we spawned the process with a list of a single Thing. We could always add a guard if we want the object to further check that it is being constructed with the proper type of arguments.

This program uses the function *self*, which retrieves the process ID of the currently running process – equivalent to Thread.currentThread in Java.

On line 35, we are instructing the process to trap the exit signal as a message. On line 47, we are matching against the standard exit message in order to destroy the object.

5.5 Generic servers

Processes which implement servers are so common in any reasonably complex application, that Erlang provides an entire module dedicated to making it easy to create a server: *gen_server*. This module features a standard interface that servers implement, plus functions to send messages to one or several servers, synchronously or asynchronously. See the reference manual for more information.

5.6 Distributed Erlang

Process IDs do not have to refer processing running on the same machine. If you start up the Erlang shell on a remote machine with a name and an optional hostname or IP address[3], you can use that *node name* to connect to that node from the local machine, spawn a process using the node name – which actually spawns the process on the remote machine – and use the returned process ID to send messages just as if it were a process on the local machine. See the reference manual for more information.

Note that the only security built in to the distributed system is a shared cookie. If you do not know a node's cookie, you cannot communicate with it. No transport-level security is provided automatically, so if you are transferring sensitive information, you will need to wrap the communcations in an SSL tunnel or use some other encryption and authentication mechanism, such as Erlang's ssh library.

If an SSL tunnel is used, then the system must be configured to reject any incoming connection not from the localhost. This helpful hint is only given because of the author's experience with corporate-mandated security mechanisms. The Erlang programmer involved in security is advised to check out the web for the latest in security for general communications and Erlang.

5.7 Synchronization not necessary

There's no such thing as synchronization in Erlang! Nor is it needed. Consider that the cases of synchronization in other languages are to prevent two threads from modifying the same global data element at the same time. Clearly in Erlang this never happens, because there is no global data.

[3]Erlang will use the host's hostname if you do not specify one. This may not be desired if your hostname is not adequate to resolve the machine on remote machines.

6 Dealing with errors

6.1 Let it fail

If the Erlang philosophy towards errors could be summarized in a slogan, it would be "Let it fail". This means that when you write code and you are about to do some defensive programming, don't do it. So, going back to function argument matching, if you want to be sure that your function will only be used for processing integers, then put in a guard to your function that checks that the argument is an integer, but *do not* write an additional function clause that handles the case for non-integers.

```
1 % DO
2
3 good(X) when is_integer(X) ->
4     X * 2.
5
6 % DO NOT
7
8 bad(X) when is_integer(X) ->
9     X * 2;
10 bad(X) ->
11     {error, "Argument to bad must be integer"}.
```

Figure 28: Good and bad error handling in matches

So what happens if you pass a non-integer to these functions? The good function will cause the process to quit with an exit reason of `{badmatch, V}`, where V is the value that caused the bad match. There will also be a stack trace so that you can see how the error came about. This is the same thing that would happen if you pass a bad argument to a Java function: an InvalidArgumentException gets thrown, and typically you don't catch those exceptions.

In the bad case, the function *returns normally*, does not cause the process to quit, and it is up to the calling function to decide what to do with the returned tuple. Pointless extra coding for everyone.

What happens if this function gets its argument from a user-supplied entry, say from a GUI? In that case, it is up to the GUI handler to validate the input, not the function. The same reasoning applies if the function gets its argument from another process. In fact, the GUI handler would be another process anyway.

6.2 Workers and supervisors

Because processes are so lightweight, do not hesitate to separate functionality into processes, even if those processes are short-lived. Thus, things that go wrong will cause a process to terminate abnormally, and it is up to the linked process to handle that error. If the linked process cannot handle a condition, it, too, should terminate abnormally, allowing a higher-level process to handle the problem.

This leads to an Erlang concept known as the *supervision tree*. The idea is that there are *worker* processes which perform actual work, and *supervisor* processes which do not perform work, but monitor workers.

Let's look at an example. In the following program, we're going to set up a server socket which will accept connections, and hand off each connection to a worker.

```
1  -module(serverexample).
2  -export([start_server/0, handle_connection/1]).
3
4  start_server() ->
5      {ok, ListenSocket} = gen_tcp:listen(8080, [binary]),
6      accept(ListenSocket).
7
8  accept(ListenSocket) ->
9      {ok, Socket} = get_tcp:accept(ListenSocket),
10     spawn(server_example, handle_connection, [Socket]),
11     accept(ListenSocket).
12
13 handle_connection(Socket) ->
14     % do stuff
15
```

Figure 29: Quick and dirty server

To start the server, we are using gen_tcp:listen, and matching it against an ok result. If there is an error, the process will terminate abnormally. Therefore, it is up to some higher-level process to handle this error. This would be a supervisor process that makes sure that the server is alive, for example by restarting it if it dies (see Figure 30).

One advantage of building a supervisor tree this way is that if the aforementioned supervisor process is terminated by a higher-level supervisor, then it will terminate its child processes before terminating, which leads to a graceful shutdown automatically.

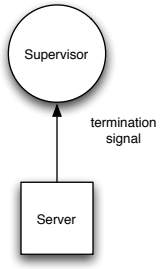

Figure 30: Supervisor and child

Suppose, instead of failing to open up the server socket, we fail to accept a new connection (by failing to match the ok condition for get_tcp:accept). In this case, we desire that the supervisor restart the server, but what happens to the children of the server process, which are presumably busy processing existing connections?

If we just cared about processing connections, we would be fine because a child process does not terminate if its parent process terminates, unless they are linked, and we did not link the processes together in the example (otherwise we would have called *spawn_link*). However, suppose we wanted the server process to also keep track of how many connections are currently being handled, so that when a new connection comes in, we increment the counter, and when an existing connection terminates, we decrement the counter.

Remember the philosophy that you should not hesitate to move functionality into a separate process. In this case we would create a process whose sole purpose is to keep track of the number of connections. It may not sound efficient, but it is, simply because Erlang processes are extremely light-weight. There is just no downside to having to increment and decrement a connection counter by passing messages to the counter process (see Figure 31).

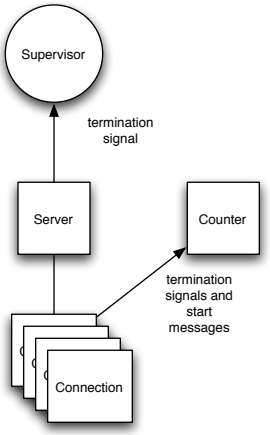

Figure 31: Added counter process

Thus, if the server dies, the counter process remains alive and maintains the correct number of outstanding connections.

If a connection terminates, the counter process needs to be notified so that it can decrease the number of connections. Because we have defined the server process as a worker process, it would be unwise to have the server process monitor the connection processes, because then it would be both a worker process and a supervisor process.

The answer could either be to write another supervisor process just for the connections, or to have the server supervisor also monitor the connection processes. Unfortunately, the standard supervisor in Erlang does not have the facility to call a user-defined function when a linked process terminates. The purpose of an Erlang supervisor is to handle termination in one of a limited set of ways.

However, we know that linking two processes will cause termination signals to be sent, and these can be handled as we saw above with process_flag and trap_exit. This could be the signal to the counter process to decrement its counter.

We have handled the conditions where the server process and the connection processes terminate abnormally (and normally) by thinking in Erlang. However, we have also added a counter process which needs to be supervised. What happens if the counter process terminates abnormally?

In this case, we know that we have lost track of the number of connections. We can use any of a number of strategies. The simplest one would be to terminate all existing connections so that

we start from a known number of connections (zero). This happens automatically, since links are bidirectional. If an unhandled exit signal is received, a process will exit. This means that if the counter process exits, and it is linked to several existing connection processes, then all of those processes will also exit.

Perhaps a more sophisticated strategy would be to have the connection processes ignore the counter's exit signal. Then, when the counter process starts up, it must find all the existing connection processes, and link to them. There is an existing server which can do this, called pg2 (or process group 2). A process can add itself to a named process group, and other processes can then query the list of processes in that group. If a member of the group terminates, it is automatically removed from the process group. See Figure 32.

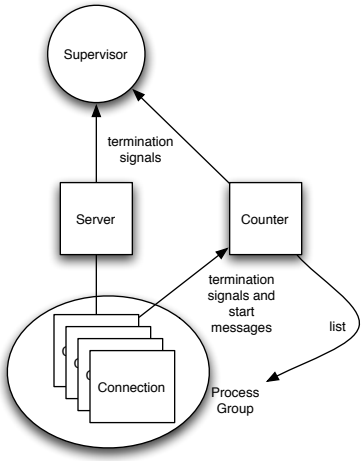

Figure 32: Counter supervision and connection process group

Thus, the counter process can query the group for the processes, and link to each one. The fun thing is that if a connection process dies after the counter process sees it in the group, but before it can link to it, then when it does link to it (incrementing the counter), it will receive an exit signal (decrementing the counter).

References

[1] http://www.erlang.org and the documentation provided therein.

[2] http://www.erlang.se and the documentation provided therein. The *Programming Rules and Conventions* is an excellent guide to writing clean Erlang code.