Erlang (programming language)

Erlang (/ˈɜːrlæŋ/ *UR-lang*) is a general-purpose, concurrent, functional programming language, and a garbage-collected runtime system. The term Erlang is used interchangeably with Erlang/OTP, or Open Telecom Platform (OTP), which consists of the Erlang runtime system, several ready-to-use components (OTP) mainly written in Erlang, and a set of design principles for Erlang programs.[3]

The Erlang runtime system is designed for systems with these traits:

- Distributed
- Fault-tolerant
- Soft real-time
- Highly available, non-stop applications
- Hot swapping, where code can be changed without stopping a system.[4]

The Erlang programming language has immutable data, pattern matching, and functional programming.[5] The sequential subset of the Erlang language supports eager evaluation, single assignment, and dynamic typing.

It was originally proprietary software within Ericsson, developed by Joe Armstrong, Robert Virding, and Mike Williams in 1986,[6] but was released as free and open-source software in 1998.[7][8] Erlang/OTP is supported and maintained by the Open Telecom Platform (OTP) product unit at Ericsson.

Contents

Erlang

Paradigms	Multi-paradigm: concurrent, functional
Designed by	Joe Armstrong Robert Virding Mike Williams
Developer	Ericsson
First appeared	1986
Stable release	23.0.3[1] / 20 July 2020
Typing discipline	Dynamic, strong
License	Apache License 2.0
Filename extensions	.erl, .hrl
Website	www.erlang.org (http://www.erlang.org)
Major implementations	
Erlang	
Influenced by	
Lisp, PLEX,[2] Prolog, Smalltalk	
Influenced	
Akka, Clojure, Dart, Elixir, F#, Opa, Oz, Reia, Rust, Scala	

Erlang Programming at Wikibooks

History

The name *Erlang*, attributed to Bjarne Däcker, has been presumed by those working on the telephony switches (for whom the language was designed) to be a reference to Danish mathematician and engineer Agner Krarup Erlang and a syllabic abbreviation of "Ericsson Language".[6][9] Erlang was designed with the aim of improving the development of telephony applications. The initial version of Erlang was implemented in Prolog and was influenced by the programming language PLEX used in earlier Ericsson exchanges. By 1988 Erlang had proven that it was suitable for prototyping telephone exchanges, but the Prolog interpreter was far too slow. One group within Ericsson estimated that it would need to be 40 times faster to be suitable for production use. In 1992, work began on the BEAM virtual machine (VM) which compiles Erlang to C using a mix of natively compiled code and threaded code to strike a balance between performance and disk space.[10] According to Armstrong, the language went from lab product to real applications following the collapse of the next-generation AXE telephone exchange named *AXE-N* in 1995. As a result, Erlang was chosen for the next asynchronous transfer mode (ATM) exchange *AXD*.[6]

In 1998 Ericsson announced the AXD301 switch, containing over a million lines of Erlang and reported to achieve a high availability of nine "9"s.[11] Shortly thereafter, Ericsson Radio Systems banned the in-house use of Erlang for new products, citing a preference for non-proprietary languages. The ban caused Armstrong and others to leave Ericsson.[12] The implementation was open-sourced at the end of the year.[6] Ericsson eventually lifted the ban and re-hired Armstrong in 2004.[12]

In 2006, native symmetric multiprocessing support was added to the runtime system and VM.[6]

Processes

Joe Armstrong, co-inventor of Erlang, summarized the principles of processes in his PhD thesis:[13]

- Everything is a process.
- Processes are strongly isolated.
- Process creation and destruction is a lightweight operation.
- Message passing is the only way for processes to interact.
- Processes have unique names.
- If you know the name of a process you can send it a message.
- Processes share no resources.
- Error handling is non-local.
- Processes do what they are supposed to do or fail.

Joe Armstrong remarked in an interview with Rackspace in 2013: "If Java is 'write once, run anywhere', then Erlang is 'write once, run forever'."[14]

Usage

In 2014, Ericsson reported Erlang was being used in its support nodes, and in GPRS, 3G and LTE mobile networks worldwide and also by Nortel and T-Mobile.[15]

As Tim Bray, director of Web Technologies at Sun Microsystems, expressed in his keynote at O'Reilly Open Source Convention (OSCON) in July 2008:

> If somebody came to me and wanted to pay me a lot of money to build a large scale message handling system that really had to be up all the time, could never afford to go down for years at a time, I would unhesitatingly choose Erlang to build it in.

Erlang is the programming language used to code WhatsApp.[16]

Since being released as open source, Erlang has been spreading beyond Telecoms, establishing itself in other verticals such as FinTech, Gaming, Healthcare, Automotive, IoT and Blockchain. Apart from WhatsApp there are other companies listed as Erlang's success stories: Vocalink (a MasterCard company), Goldman Sachs, Nintendo, AdRoll, Grindr, BT Mobile, Samsung, OpenX, SITA.[17][18]

Functional programming examples

Factorial

A factorial algorithm implemented in Erlang:

```
-module(fact). % This is the file 'fact.erl', the module and the filename must match
-export([fac/1]). % This exports the function 'fac' of arity 1 (1 parameter, no type, no name)

fac(0) -> 1; % If 0, then return 1, otherwise (note the semicolon ; meaning 'else')
fac(N) when N > 0, is_integer(N) -> N * fac(N-1).
% Recursively determine, then return the result
% (note the period . meaning 'endif' or 'function end')
%% This function will crash if anything other than a nonnegative integer is given.
%% It illustrates the "Let it crash" philosophy of Erlang.
```

Fibonacci sequence

A tail recursive algorithm that produces the Fibonacci sequence:

```
%% The module declaration must match the file name "series.erl"
-module(series).

%% The export statement contains a list of all those functions that form
%% the module's public API.  In this case, this module exposes a single
%% function called fib that takes 1 argument (I.E. has an arity of 1)
%% The general syntax for -export is a list containing the name and
%% arity of each public function
-export([fib/1]).

%% ------------------------------------------------------------------
%% Public API
%% ------------------------------------------------------------------

%% Handle cases in which fib/1 receives specific values
%% The order in which these function signatures are declared is a vital
%% part of this module's functionality
```

```
%% If fib/1 is passed precisely the integer 0, then return 0
fib(0) -> 0;

%% If fib/1 receives a negative number, then return the atom err_neg_val
%% Normally, such defensive coding is discouraged due to Erlang's 'Let
%% it Crash' philosophy; however, in this case we should explicitly
%% prevent a situation that will crash Erlang's runtime engine
fib(N) when N < 0 -> err_neg_val;

%% If fib/1 is passed an integer less than 3, then return 1
%% The preceding two function signatures handle all cases where N < 1,
%% so this function signature handles cases where N = 1 or N = 2
fib(N) when N < 3 -> 1;

%% For all other values, call the private function fib_int/3 to perform
%% the calculation
fib(N) -> fib_int(N, 0, 1).

%% --------------------------------------------------------------------
%% Private API
%% --------------------------------------------------------------------

%% If fib_int/3 receives a 1 as its first argument, then we're done, so
%% return the value in argument B.  Since we are not interested in the
%% value of the second argument, we denote this using _ to indicate a
%% "don't care" value
fib_int(1, _, B) -> B;

%% For all other argument combinations, recursively call fib_int/3
%% where each call does the following:
%%  - decrement counter N
%%  - Take the previous fibonacci value in argument B and pass it as
%%    argument A
%%  - Calculate the value of the current fibonacci number and pass it
%%    as argument B
fib_int(N, A, B) -> fib_int(N-1, B, A+B).
```

Here's the same program without the explanatory comments:

```
-module(series).
-export([fib/1]).

fib(0) -> 0;
fib(N) when N < 0 -> err_neg_val;
fib(N) when N < 3 -> 1;
fib(N) -> fib_int(N, 0, 1).

fib_int(1, _, B) -> B;
fib_int(N, A, B) -> fib_int(N-1, B, A+B).
```

Quicksort

Quicksort in Erlang, using list comprehension:[19]

```
%% qsort:qsort(List)
%% Sort a list of items
-module(qsort).     % This is the file 'qsort.erl'
-export([qsort/1]). % A function 'qsort' with 1 parameter is exported (no type, no name)

qsort([]) -> []; % If the list [] is empty, return an empty list (nothing to sort)
qsort([Pivot|Rest]) ->
    % Compose recursively a list with 'Front' for all elements that should be before 'Pivot'
    % then 'Pivot' then 'Back' for all elements that should be after 'Pivot'
    qsort([Front || Front <- Rest, Front < Pivot]) ++
    [Pivot] ++
    qsort([Back || Back <- Rest, Back >= Pivot]).
```

The above example recursively invokes the function `qsort` until nothing remains to be sorted. The expression `[Front || Front <- Rest, Front < Pivot]` is a list comprehension, meaning "Construct a list of elements `Front` such that `Front` is a member of `Rest`, and `Front` is less than

`Pivot`." `++` is the list concatenation operator.

A comparison function can be used for more complicated structures for the sake of readability.

The following code would sort lists according to length:

```
% This is file 'listsort.erl' (the compiler is made this way)
-module(listsort).
% Export 'by_length' with 1 parameter (don't care about the type and name)
-export([by_length/1]).

by_length(Lists) -> % Use 'qsort/2' and provides an anonymous function as a parameter
   qsort(Lists, fun(A,B) -> length(A) < length(B) end).

qsort([], _)-> []; % If list is empty, return an empty list (ignore the second parameter)
qsort([Pivot|Rest], Smaller) ->
    % Partition list with 'Smaller' elements in front of 'Pivot' and not-'Smaller' elements
    % after 'Pivot' and sort the sublists.
    qsort([X || X <- Rest, Smaller(X,Pivot)], Smaller)
    ++ [Pivot] ++
    qsort([Y || Y <- Rest, not(Smaller(Y, Pivot))], Smaller).
```

A `Pivot` is taken from the first parameter given to `qsort()` and the rest of `Lists` is named `Rest`. Note that the expression

```
[X || X <- Rest, Smaller(X,Pivot)]
```

is no different in form from

```
[Front || Front <- Rest, Front < Pivot]
```

(in the previous example) except for the use of a comparison function in the last part, saying "Construct a list of elements `X` such that `X` is a member of `Rest`, and `Smaller` is true", with `Smaller` being defined earlier as

```
fun(A,B) -> length(A) < length(B) end
```

The anonymous function is named `Smaller` in the parameter list of the second definition of `qsort` so that it can be referenced by that name within that function. It is not named in the first definition of `qsort`, which deals with the base case of an empty list and thus has no need of this function, let alone a name for it.

Data types

Erlang has eight primitive data types:

Integers
: Integers are written as sequences of decimal digits, for example, 12, 12375 and -23427 are integers. Integer arithmetic is exact and only limited by available memory on the machine. (This is called arbitrary-precision arithmetic.)

Atoms
: Atoms are used within a program to denote distinguished values. They are written as strings of consecutive alphanumeric characters, the first character being lowercase. Atoms can contain any character if they are enclosed within single quotes and an escape convention exists which allows any character to be used within an atom. Atoms are never garbage collected and should be used with caution, especially if using dynamic atom generation.

Floats
: Floating point numbers use the IEEE 754 64-bit representation.

References
: References are globally unique symbols whose only property is that they can be compared for equality. They are created by evaluating the Erlang primitive `make_ref()`.

Binaries
: A binary is a sequence of bytes. Binaries provide a space-efficient way of storing binary data. Erlang primitives exist for composing and decomposing binaries and for efficient input/output of binaries.

Pids
: Pid is short for *process identifier* – a Pid is created by the Erlang primitive `spawn(...)` Pids are references to Erlang processes.

Ports
: Ports are used to communicate with the external world. Ports are created with the built-in function `open_port`. Messages can be sent to and received from ports, but these messages must obey the so-called "port protocol."

Funs
: Funs are function closures. Funs are created by expressions of the form: `fun(...) -> ... end`.

And three compound data types:

Tuples
: Tuples are containers for a fixed number of Erlang data types. The syntax `{D1,D2,...,Dn}` denotes a tuple whose arguments are `D1, D2, ... Dn`. The arguments can be primitive data types or compound data types. Any element of a tuple can be accessed in constant time.

Lists
: Lists are containers for a variable number of Erlang data types. The syntax `[Dh|Dt]` denotes a list whose first element is `Dh`, and whose remaining elements are the list `Dt`. The syntax `[]` denotes an empty list. The syntax `[D1,D2,..,Dn]` is short for `[D1|[D2|..|[Dn|[]]]]`. The first element of a list can be accessed in constant time. The first element of a list is called the *head* of the list. The remainder of a list when its head has been removed is called the *tail* of the list.

Maps
: Maps contain a variable number of key-value associations. The syntax is `#{Key1=>Value1,...,KeyN=>ValueN}`.

Two forms of syntactic sugar are provided:

Strings
: Strings are written as doubly quoted lists of characters. This is syntactic sugar for a list of the integer Unicode code points for the characters in the string. Thus, for example, the string "cat" is shorthand for `[99,97,116]`.[20]

Records
: Records provide a convenient way for associating a tag with each of the elements in a tuple. This allows one to refer to an element of a tuple by name and not by position. A pre-compiler takes the record definition and replaces it with the appropriate tuple reference.

Erlang has no method to define classes, although there are external libraries available.[21]

"Let it Crash" coding style

In most other programming languages, software crashes have always been (and often still are) considered highly undesirable situations that must be avoided at all costs. Consequently, elaborate exception handling mechanisms exist to trap these situations and then mitigate their effects. This design philosophy exists because many of the foundational principles of software design were defined at a time when computers were single processor machines. Under these conditions, software crashes were indeed fatal. Given this basic constraint, it was perfectly natural therefore to develop programming styles in which a large proportion of the code was dedicated to detecting and then handling error situations. This in turn, led directly to the still widely popular coding style known as defensive programming.

However, the designers of Erlang realised that in spite of their undesirable effects, software crashes are much like death and taxes - quite unavoidable. Therefore, rather than treating a crash as a crisis situation that temporarily suspends all normal work until a solution is found, they reasoned it would make far greater sense to treat a crash in exactly the same manner as any other normal runtime event. Consequently, when an Erlang process crashes, this situation is reported as just another type of message arriving in a process' mail box.

This realisation led to Erlang's designers building a language with the following fundamental features:

- Erlang has no concept of global memory; therefore relative to each other, all processes are isolated execution environments
- Erlang processes can:
  - be spawned very cheaply
  - only communicate using message passing
  - monitor each other. This allows processes to be arranged in hierarchies known as "supervisor trees."
- A process should perform its task or fail
- Process failure is reported simply as a message

The "Let it Crash" style of coding is therefore the practical consequence of working in a language that operates on these principles.

Supervisor trees

A typical Erlang application is written in the form of a supervisor tree. This architecture is based on a hierarchy of processes in which the top level process is known as a "supervisor". The supervisor then spawns multiple child processes that act either as workers or more, lower level supervisors. Such hierarchies can exist to arbitrary depths and have proven to provide a highly scalable and fault-tolerant environment within which application functionality can be implemented.

Within a supervisor tree, all supervisor processes are responsible for managing the lifecycle of their child processes, and this includes handling situations in which those child processes crash. Any process can become a supervisor by first spawning a child process, then calling `erlang:monitor/2` on that process. If the monitored process then crashes, the supervisor will receive a message containing a tuple whose first member is the atom `'DOWN'`. The supervisor is responsible firstly for listening for such messages and secondly, for taking the appropriate action to correct the error condition.

In addition, "Let it Crash" results in a style of coding that contains little defensive code, resulting in smaller applications.

Concurrency and distribution orientation

Erlang's main strength is support for concurrency. It has a small but powerful set of primitives to create processes and communicate among them. Erlang is conceptually similar to the language occam, though it recasts the ideas of communicating sequential processes (CSP) in a functional framework and uses asynchronous message passing.[22] Processes are the primary means to structure an Erlang application. They are neither operating system processes nor threads, but lightweight processes that are scheduled by BEAM. Like operating system processes (but unlike operating system threads), they share no state with each other. The estimated minimal overhead for each is 300 words.[23] Thus, many processes can be created without degrading performance. In 2005, a benchmark with 20 million processes was successfully performed with 64-bit Erlang on a machine with 16 GB random-access memory (RAM; total 800 bytes/process).[24] Erlang has supported symmetric multiprocessing since release R11B of May 2006.

While threads need external library support in most languages, Erlang provides language-level features to create and manage processes with the goal of simplifying concurrent programming. Though all concurrency is explicit in Erlang, processes communicate using message passing instead of shared variables, which removes the need for explicit locks (a locking scheme is still used internally by the VM).[25]

Inter-process communication works via a shared-nothing asynchronous message passing system: every process has a "mailbox", a queue of messages that have been sent by other processes and not yet consumed. A process uses the `receive` primitive to retrieve messages that match desired patterns. A message-handling routine tests messages in turn against each pattern, until one of them matches. When the message is consumed and removed from the mailbox the process resumes execution. A message may comprise any Erlang structure, including primitives (integers, floats, characters, atoms), tuples, lists, and functions.

The code example below shows the built-in support for distributed processes:

```
% Create a process and invoke the function web:start_server(Port, MaxConnections)
ServerProcess = spawn(web, start_server, [Port, MaxConnections]),

% Create a remote process and invoke the function
% web:start_server(Port, MaxConnections) on machine RemoteNode
RemoteProcess = spawn(RemoteNode, web, start_server, [Port, MaxConnections]),

% Send a message to ServerProcess (asynchronously). The message consists of a tuple
% with the atom "pause" and the number "10".
ServerProcess ! {pause, 10},

% Receive messages sent to this process
receive
        a_message -> do_something;
        {data, DataContent} -> handle(DataContent);
        {hello, Text} -> io:format("Got hello message: ~s", [Text]);
        {goodbye, Text} -> io:format("Got goodbye message: ~s", [Text])
end.
```

As the example shows, processes may be created on remote nodes, and communication with them is transparent in the sense that communication with remote processes works exactly as communication with local processes.

Concurrency supports the primary method of error-handling in Erlang. When a process crashes, it neatly exits and sends a message to the controlling process which can then take action, such as for instance starting a new process that takes over the old process's task.[26][27]

Implementation

The official reference implementation of Erlang uses BEAM.[28] BEAM is included in the official distribution of Erlang, called Erlang/OTP. BEAM executes bytecode which is converted to threaded code at load time. It also includes a native code compiler on most platforms, developed by the High Performance Erlang Project

(HiPE) at Uppsala University. Since October 2001 the HiPE system is fully integrated in Ericsson's Open Source Erlang/OTP system.[29] It also supports interpreting, directly from source code via abstract syntax tree, via script as of R11B-5 release of Erlang.

Hot code loading and modules

Erlang supports language-level Dynamic Software Updating. To implement this, code is loaded and managed as "module" units; the module is a compilation unit. The system can keep two versions of a module in memory at the same time, and processes can concurrently run code from each. The versions are referred to as the "new" and the "old" version. A process will not move into the new version until it makes an external call to its module.

An example of the mechanism of hot code loading:

```
%% A process whose only job is to keep a counter.
%% First version
-module(counter).
-export([start/0, codeswitch/1]).

start() -> loop(0).

loop(Sum) ->
  receive
     {increment, Count} ->
        loop(Sum+Count);
     {counter, Pid} ->
        Pid ! {counter, Sum},
        loop(Sum);
     code_switch ->
        ?MODULE:codeswitch(Sum)
        % Force the use of 'codeswitch/1' from the latest MODULE version
  end.

codeswitch(Sum) -> loop(Sum).
```

For the second version, we add the possibility to reset the count to zero.

```
%% Second version
-module(counter).
-export([start/0, codeswitch/1]).

start() -> loop(0).

loop(Sum) ->
  receive
     {increment, Count} ->
        loop(Sum+Count);
     reset ->
        loop(0);
     {counter, Pid} ->
        Pid ! {counter, Sum},
        loop(Sum);
     code_switch ->
        ?MODULE:codeswitch(Sum)
  end.

codeswitch(Sum) -> loop(Sum).
```

Only when receiving a message consisting of the atom `code_switch` will the loop execute an external call to codeswitch/1 (`?MODULE` is a preprocessor macro for the current module). If there is a new version of the *counter* module in memory, then its codeswitch/1 function will be called. The practice of having a specific entry-point into a new version allows the programmer to transform state to what is needed in the newer version. In the example, the state is kept as an integer.

In practice, systems are built up using design principles from the Open Telecom Platform, which leads to more code upgradable designs. Successful hot code loading is exacting. Code must be written with care to make use of Erlang's facilities.

Distribution

In 1998, Ericsson released Erlang as free and open-source software to ensure its independence from a single vendor and to increase awareness of the language. Erlang, together with libraries and the real-time distributed database Mnesia, forms the OTP collection of libraries. Ericsson and a few other companies support Erlang commercially.

Since the open source release, Erlang has been used by several firms worldwide, including Nortel and T-Mobile.[30] Although Erlang was designed to fill a niche and has remained an obscure language for most of its existence, its popularity is growing due to demand for concurrent services.[31][32] Erlang has found some use in fielding massively multiplayer online role-playing game (MMORPG) servers.[33]

See also

- Elixir – a functional, concurrent, general-purpose programming language that runs on BEAM
- Lisp Flavored Erlang (LFE) – a Lisp-based programming language that runs on BEAM
- Mix (build tool)
- Phoenix (web framework)

References

1. "Releases - erlang/otp" (https://github.com/erlang/otp/releases). Retrieved 4 August 2020 – via GitHub.
2. Conferences, N. D. C. (4 June 2014). "Joe Armstrong - Functional Programming the Long Road to Enlightenment: a Historical and Personal Narrative" (https://vimeo.com/97329186). Vimeo.
3. "Erlang – Introduction" (http://erlang.org/doc/system_architecture_intro/sys_arch_intro.html#id58791). *erlang.org*.
4. Armstrong, Joe; Däcker, Bjarne; Lindgren, Thomas; Millroth, Håkan. "Open-source Erlang – White Paper" (https://web.archive.org/web/20111025022940/http://ftp.sunet.se/pub/lang/erlang/white_paper.html). Archived from the original (http://ftp.sunet.se/pub/lang/erlang/white_paper.html) on 25 October 2011. Retrieved 31 July 2011.
5. Hitchhiker's Tour of the BEAM – Robert Virding http://www.erlang-factory.com/upload/presentations/708/HitchhikersTouroftheBEAM.pdf
6. Armstrong, Joe (2007). *History of Erlang*. *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*. ISBN 978-1-59593-766-7.
7. "How tech giants spread open source programming love - CIO.com" (http://www.cio.com/article/3020454/open-source-tools/how-tech-giants-spread-open-source-programming-love.html).
8. "Erlang/OTP Released as Open Source, 1998-12-08" (https://web.archive.org/web/19991009002753/http://www.erlang.se/onlinenews/ErlangOTPos.shtml). Archived from the original (http://www.erlang.se/onlinenews/ErlangOTPos.shtml) on 9 October 1999.
9. "Erlang, the mathematician?" (http://www.erlang.org/pipermail/erlang-questions/1999-February/000098.html).
10. Armstrong, Joe (August 1997). "The development of Erlang". *ACM SIGPLAN Notices*. **32** (8): 196–203. doi:10.1145/258948.258967 (https://doi.org/10.1145%2F258948.258967). ISBN 0897919181.

11. "Concurrency Oriented Programming in Erlang" (http://www.rabbitmq.com/resources/armstrong.pdf) (PDF). 9 November 2002.
12. "question about Erlang's future" (http://erlang.org/pipermail/erlang-questions/2006-July/021368.html). 6 July 2010.
13. http://erlang.org/download/armstrong_thesis_2003.pdf
14. McGreggor, Duncan (26 March 2013). *Rackspace takes a look at the Erlang programming language for distributed computing* (https://www.youtube.com/watch?v=u41GEwIq2mE&t=3m59s) (Video). Rackspace Studios, SFO. Retrieved 24 April 2019.
15. "Ericsson" (http://www.ericsson.com/news/141204-inside-erlang-creator-joe-armstrong-tells-his-story_244099435_c). *Ericsson.com*. 4 December 2014. Retrieved 7 April 2018.
16. "Inside Erlang, The Rare Programming Language Behind WhatsApp's Success" (https://www.fastcompany.com/3026758/inside-erlang-the-rare-programming-language-behind-whatsapps-success). *fastcompany.com*. 21 February 2014. Retrieved 12 November 2019.
17. "Which companies are using Erlang, and why? #MyTopdogStatus" (https://www.erlang-solutions.com/blog/which-companies-are-using-erlang-and-why-mytopdogstatus.html). *erlang-solutions.com*. 11 September 2019. Retrieved 15 March 2020.
18. "Which new companies are using Erlang and Elixir? #MyTopdogStatus" (https://www.erlang-solutions.com/blog/which-new-companies-are-using-erlang-and-elixir-mytopdogstatus.html). *erlang-solutions.com*. 2 March 2020. Retrieved 24 June 2020.
19. "Erlang – List Comprehensions" (http://erlang.org/doc/programming_examples/list_comprehensions.html). *erlang.org*.
20. "String and Character Literals" (http://erlang.org/doc/apps/stdlib/unicode_usage.html#string-and-character-literals). Retrieved 2 May 2015.
21. "ect – Erlang Class Transformation – add object-oriented programming to Erlang – Google Project Hosting" (https://code.google.com/p/ect/). Retrieved 2 May 2015.
22. Armstrong, Joe (September 2010). "Erlang" (https://doi.org/10.1145/1810891.1810910). *Communications of the ACM*. **53** (9): 68–75. doi:10.1145/1810891.1810910 (https://doi.org/10.1145%2F1810891.1810910). "Erlang is conceptually similar to the occam programming language, though it recasts the ideas of CSP in a functional framework and uses asynchronous message passing."
23. "Erlang Efficiency Guide – Processes" (https://web.archive.org/web/20150227174813/http://www.erlang.org/doc/efficiency_guide/processes.html). Archived from the original (http://www.erlang.org/doc/efficiency_guide/processes.html) on 27 February 2015.
24. Wiger, Ulf (14 November 2005). "Stress-testing erlang" (https://groups.google.com/group/comp.lang.functional/msg/33b7a62afb727a4f?dmode=source). *comp.lang.functional.misc*. Retrieved 25 August 2006.
25. "Lock-free message queue" (http://erlang.2086793.n4.nabble.com/Lock-free-message-queue-td2550221.html). Retrieved 23 December 2013.
26. Armstrong, Joe. "Erlang robustness" (https://web.archive.org/web/20150423182840/http://www.erlang.org/doc/getting_started/robustness.html). Archived from the original (http://www.erlang.org/doc/getting_started/robustness.html) on 23 April 2015. Retrieved 15 July 2010.
27. "Erlang Supervision principles" (https://web.archive.org/web/20150206050600/http://www.erlang.org/doc/design_principles/sup_princ.html). Archived from the original (http://www.erlang.org/doc/design_principles/sup_princ.html) on 6 February 2015. Retrieved 15 July 2010.
28. "Erlang – Compilation and Code Loading" (http://erlang.org/doc/reference_manual/code_loading.html#id90080). *erlang.org*. Retrieved 21 December 2017.
29. "High Performance Erlang" (http://www.it.uu.se/research/group/hipe/). Retrieved 26 March 2011.

30. "Who uses Erlang for product development?" (http://erlang.org/faq/introduction.html#idp32141008). *Frequently asked questions about Erlang*. Retrieved 16 July 2007. "*The largest user of Erlang is (surprise!) Ericsson. Ericsson use it to write software used in telecommunications systems. Many dozens of projects have used it, a particularly large one is the extremely scalable AXD301 ATM switch.* Other commercial users listed as part of the FAQ include: Nortel, Deutsche Flugsicherung (the German national air traffic control organisation), and T-Mobile."
31. "Programming Erlang" (http://www.ddj.com/linux-open-source/201001928?cid=RSSfeed_DDJ_OpenSource). Retrieved 13 December 2008. "Virtually all language use shared state concurrency. This is very difficult and leads to terrible problems when you handle failure and scale up the system...Some pretty fast-moving startups in the financial world have latched onto Erlang; for example, the Swedish www.kreditor.se."
32. "Erlang, the next Java" (https://web.archive.org/web/20071011065959/http://cincomsmalltalk.com/userblogs/ralph/blogView?showComments=true&entry=3364027251). Archived from the original (http://www.cincomsmalltalk.com/userblogs/ralph/blogView?showComments=true&entry=3364027251) on 11 October 2007. Retrieved 8 October 2008. "I do not believe that other languages can catch up with Erlang anytime soon. It will be easy for them to add language features to be like Erlang. It will take a long time for them to build such a high-quality VM and the mature libraries for concurrency and reliability. So, Erlang is poised for success. If you want to build a multicore application in the next few years, you should look at Erlang."
33. Clarke, Gavin (5 February 2011). "Battlestar Galactica vets needed for online roleplay" (https://www.theregister.co.uk/2011/02/05/battlestar_galactica_mmp/). *Music and Media*. The Reg. Retrieved 8 February 2011.

Further reading

- Armstrong, Joe (2003). "Making reliable distributed systems in the presence of software errors" (https://web.archive.org/web/20150323001245/https://www.sics.se/~joe/thesis/armstrong_thesis_2003.pdf) (PDF). PhD Dissertation. The Royal Institute of Technology, Stockholm, Sweden. Archived from the original on 23 March 2015. Retrieved 13 February 2016.
- Armstrong, Joe (2007). "A history of Erlang". *Proceedings of the third ACM SIGPLAN conference on History of programming languages – HOPL III*. pp. 6–1. doi:10.1145/1238844.1238850 (https://doi.org/10.1145%2F1238844.1238850). ISBN 978-1-59593-766-7.
- Early history of Erlang (http://www.erlang.se/publications/bjarnelic.pdf) by Bjarne Däcker
- Mattsson, H.; Nilsson, H.; Wikstrom, C. (1999). "Mnesia – A distributed robust DBMS for telecommunications applications". *First International Workshop on Practical Aspects of Declarative Languages (PADL '99)*: 152–163.
- Armstrong, Joe; Virding, Robert; Williams, Mike; Wikstrom, Claes (16 January 1996). *Concurrent Programming in Erlang* (https://web.archive.org/web/20120306002430/http://www.erlang.org/erlang_book_toc.html) (2nd ed.). Prentice Hall. p. 358. ISBN 978-0-13-508301-7. Archived from the original (http://www.erlang.org/erlang_book_toc.html) on 6 March 2012.
- Armstrong, Joe (11 July 2007). *Programming Erlang: Software for a Concurrent World* (https://archive.org/details/programmingerlan0000arms/page/536) (1st ed.). Pragmatic Bookshelf. p. 536 (https://archive.org/details/programmingerlan0000arms/page/536). ISBN 978-1-934356-00-5.
- Thompson, Simon J.; Cesarini, Francesco (19 June 2009). *Erlang Programming: A Concurrent Approach to Software Development* (http://www.erlangprogramming.org) (1st ed.). Sebastopol, California: O'Reilly Media, Inc. p. 496. ISBN 978-0-596-51818-9.
- Logan, Martin; Merritt, Eric; Carlsson, Richard (28 May 2010). *Erlang and OTP in Action* (1st ed.). Greenwich, CT: Manning Publications. p. 500. ISBN 978-1-933988-78-8.
- Martin, Brown (10 May 2011). "Introduction to programming in Erlang, Part 1: The basics" (http://www.ibm.com/developerworks/opensource/library/os-erlang1/index.html). *developerWorks*. IBM. Retrieved 10 May 2011.

- Martin, Brown (17 May 2011). "Introduction to programming in Erlang, Part 2: Use advanced features and functionality" (http://www.ibm.com/developerworks/opensource/library/os-erlang2/index.html). *developerWorks*. IBM. Retrieved 17 May 2011.
- Wiger, Ulf (30 March 2001). "Four-fold Increase in Productivity and Quality: Industrial-Strength Functional Programming in Telecom-Class Products" (http://www.erlang.se/publications/Ulf_Wiger.pdf) (PDF). *FEmSYS 2001 Deployment on distributed architectures*. Ericsson Telecom AB. Retrieved 16 September 2014.

External links

- Official website (https://www.erlang.org/)
- "Inside Erlang – Creator Joe Armstrong Tells His Story" (http://www.ericsson.com/news/141204-inside-erlang-creator-joe-armstrong-tells-his-story_244099435_c). *Ericsson*. 4 December 2014.