Function Clause

- This is a **function clause**:

```
name(arguments) -> expr1, expr2, ... exprN
```

- Expressions separated by commas – comma means "and"

- Value of last expression `exprN` is value of the clause

- (Every Erlang expression produces a value)

- "`name(arguments)`" is **head** of the clause

Example

```
arith(X, Y) ->
    io:fwrite("Arguments: ~p ~p~n", [ X, Y ]) ,
    Sum = X + Y ,
    Diff = X - Y ,
    Prod = X * Y ,
    Quo = X div Y ,
    io:fwrite("~p ~p ~p ~p~n", [ Sum, Diff, Prod, Quo ]) ,
    { Sum, Diff, Prod, Quo } .
```

Note:

- Function name starts with lowercase letter
- `io:fwrite` similar to `printf`
- Variables start with capital letter
- Expressions separated by comma
- Clause ended by period
- Final expression is function's return value

Function Definition

- Function may have SEVERAL heads ... function is SEQUENCE OF pattern matching clauses separated by semicolons – semicolon means “or”

- Function head seeks to match call arguments to pattern in some head

Example:

```
what_day(saturday) -> // "saturday" is an atom
    weekend ;         // notice semicolon
what_day(sunday) ->   // "sunday" is an atom
    weekend ;         // semicolon again
what_day(_) ->        // underscore is "don’t care" variable
    weekday .         // period ends function
```

Function Examples, I

In file `example.erl`:

```
drivers_license(Age) when Age < 16 ->
    forbidden ;
drivers_license(Age) when Age == 16 ->
    'learners permit' ;
drivers_license(Age) when Age == 17 ->
    'probationary license' ;
drivers_license(Age) when Age >= 65 ->
    'vision test recommended but not required' ;
drivers_license(_) ->
    'full license'.
```

- "`when ...`" is a **clause guard**

- Clause matches if function name, arguments, and all guards match the input

Function Examples, II

```
$ erl
Erlang R14B04 ...

Eshell V5.8.5  (abort with ^G)
1> c(example).                // c() compiles
{ok,example}
2> drivers_license(16).       // must specify module
** exception error: undefined shell command drivers_license/1
3> example:drivers_license(16).
'learners permit'
4> example:drivers_license(15).
forbidden
5> example:drivers_license(17).
'probationary license'
6> example:drivers_license(23).
'full license'
7> example:drivers_license(65).
'vision test recommended but not required'
8> q().
ok
```

Function Call

Except for "built-in functions," must specify function's module when calling

```
2> drivers_license(16).
** exception error: undefined shell command drivers_license/1
3> example:drivers_license(16).
'learners permit'
```

Much-used modules in Erlang library:

`io`, `list`, `dict`, `sets`, `gb_trees`

Comments

```
%  comment begins with %, convention is to use two

%%  value returned by function is X+Y
add(X, Y) ->
    X + Y.


%%  try to pattern-match argument to each successive clause
%%  notice semicolon separator ... semicolon means "or"
abs_value(X) when X >= 0 ->
    X;
abs_value(X) ->
    -X.


%%  tuple returned by function contains both roots
%%  notice comma separator ... comma means "and"
both_sqrt(X) ->
    Pos = math:sqrt(X),
    Neg = -Pos,
    {Pos, Neg}.
```

Module Definition

- Definition: in file "`foo.erl`"

```
-module(foo).
```

- As in Java, module name and file name must match

- Also:

```
-import(module, [function/arity, function/arity, ...]).

-export([function/arity, function/arity, ...]).
```

- `export_all` compiler flag useful during debugging:

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

I/O

- I/O functions in module "`io`" (read about it at `http://erlang.org/doc/man/io.html`)

- Function "`fwrite`" (or "`format`") similar to `printf`

- However ... format characters begin with tilde (~) rather than percent

- "Pretty print" format character ~`p` knows how to print many types – use it!

- `fread` gets input from stdin

I/O Example

```
iotest() ->
    {ok, [Num, Str]} = io:fread("integer & string please: ", "~d~s"),
    io:fwrite("Num = ~p, Str = ~s~n", [Num, Str]).


%    {ok, [Num, Str]} = io:fread("integer & string please: ", "~d~s"),
%
%%   first argument: prompt string
%%   second argument: format string indicates type of input value(s)
%%   return value: 2-part tuple
%%       first part is "ok" if read operation succeeded
%%       second part is list with as many variables as input values

%    io:fwrite("Num = ~p, Str = ~s~n", [Num, Str]).
%
%%   first argument: format string including control characters
%%                   that indicate type of each output value
%%   second argument: list of output values
```

I/O Gotcha

- Erlang is dynamically typed: types checked at runtime, not compile time

- `fwrite` call will compile without error/warning even if 2nd arg is not a list

- This will compile then crash:

```
io:fwrite("Num = ~p~n", Num).
```

- 2nd arg must be a list:

```
io:fwrite("Num = ~p~n", [Num]).
```

If Statement, I

`if` uses pattern matching:

```
iftest1(N) ->
    if N < 100 ->
        io:fwrite("less than 100~n") ;
    true ->
        io:fwrite("greater than or equal to 100~n")
    end .
```

`true` atom serves as "else" case

- (Notice: semicolon & comma are not expression TERMINATORS, they are expression SEPARATORS – C/Java programmers will make lots of syntax errors)

If Statement, II

- This code

```
iftest2(N) ->
    if N < 100 ->
        io:fwrite("less than 100~n") ;
    end .
```

will crash with argument 100

- Error message: "`no true branch found when evaluating an if expression`"

- This is consequence of fact that every expression must pattern-match to a value

- Erlang forces programmer to cover all cases!

Interesting Features

- No loops

- No global or shared variables

- Like Java, Erlang uses a virtual machine – compiles "`.erl`" source code into "`.beam`" bytecodes

- ("Erjang" is implementation of Erlang on JVM)

