リスト 1: d01.erl

```
%% d01.erl

%% 再帰関数

-module(d01). % モジュールの宣言
-export([fact/1, append/2]). % エクスポートする関数の宣言

%% 階乗
fact(0) ->
  1; % Prologerはセミコロンである点に注意
fact(N) when N > 0 -> % when ... はガード
  N * fact(N -1).

%% リストの連接
append([], List) ->
  List;
append([First|Rest], List) ->
  [First|append(Rest, List)].
```

リスト 2: d02.erl

```
%% d02.erl

%% 高階関数、ラムダ式

-module(d02).
-compile(export_all). % コンパイラに全部エクスポートするように指令

sumup(From, To, Fun) -> % 第3引数にfun
  lists:sum(
    lists:map(Fun, lists:seq(From, To))).

make_inc(N) -> % 戻り値がfun
  fun (X) -> X + N end. % '->' を忘れるのだ（檜山だけ？）

sq(N) -> % テスト用、平方する関数
  N * N.
```

リスト 3: d03.erl

```
%% d03.erl

%% パターンマッチ

-module(d03).
-compile(export_all).

p1({Name, Age}) -> % 既にお馴染み、引数にパターン
  io:format("Your name: ~s~n", [Name]), % カンマは順次実行
  io:format("Your age: ~p~n", [Age]).

p2({Name, Age}) when is_list(Name), is_number(Age) -> % ガードで制約をきつく
  io:format("Your name: ~s~n", [Name]),
  io:format("Your age: ~p~n", [Age]);
p2(Name) when is_list(Name) -> % 別なパターン＆ガード
  io:format("Your name: ~p~n", [Name]).

p3(X) -> % p3と同じ、case式はもっともよく使われる制御構造
  case X of
    Name when is_list(Name) ->
      io:format("Your name: ~s~n", [Name]);
```

```
    {Name, Age} when is_list(Name), is_number(Age) ->
      io:format("Your name: ~s~n", [Name]),
      io:format("Your age: ~p~n", [Age])
  end.
```

リスト 4: d04.erl

```
%% d04.erl

%% プロセスとメッセージ

-module(d04).
-compile(export_all).

start() -> % startって名前は慣例的
  spawn(fun main/0).

main() ->
  receive
    p -> % print
      io:format("Hello~n"),
      main(); % 末尾再帰
    s -> % stop
      io:format("stop.~n"),
      ok; % プロセスも自然終了
    f -> % flush
      flush(),
      main() % 末尾再帰
  end.

flush() ->
  receive
  Any -> % 何かあれば
      io:format("flushing ~p~n", [Any]), % []をよく忘れる（檜山だけ？）
      flush() % 繰り返し
  after 0 -> % 何もないなら
      io:format("done.~n") % 抜ける、戻り値はなんだっていい
  end.
```

リスト 5: d05.erl

```
%% d05.erl

%% 自発的に動き続けるプロセス

-module(d05).
-compile(export_all).

start() ->
  spawn(fun main/0).

main() ->
  receive
    b -> % break
      break(); % 再帰じゃないけど末尾呼び出し (last call)
    s -> % stop
      io:format("stop.~n"); % プロセスも自然終了
    _Other ->
      main() % キューのフラッシュ
  after 500 -> % 0.5秒ごとに
      io:format("Hello.~n"),
      main() % 末尾再帰
  end.
```

```
23
24 break() ->
25   receive
26   c -> % continue
27       main(); % これでmainに戻る
28   _Other ->
29       io:format("break and stop.~n") % 終わり
30   end.
```

リスト 6: Counter.java

```
1  /* Counter.java */
2  public class Counter {
3    private int count; // 内部状態
4
5    public Counter(int init) {
6      count = init;
7    }
8    public void up() {
9      count++; // 状態の変更
10   }
11   public void down() {
12     count--; // 状態の変更
13   }
14   public int value() {
15     return count; // 問い合わせに応える
16   }
17 }
```

リスト 7: counter.erl

```
1  %% counter.erl
2
3  %% カウンタ
4
5  -module(counter).
6  -compile(export_all). % お行儀悪い
7
8  %% new Constructor(init) 相当
9  start(Init) ->
10   spawn(?MODULE, main, [Init]).
11
12 start(Name, Init) ->
13   register(Name, spawn(?MODULE, main, [Init])).
14
15 main(Count) ->
16   receive
17     up -> % up() 相当
18       main(Count + 1);
19     down -> % down() 相当
20       main(Count - 1);
21     {value, Pid} -> % value() 相当
22       Pid ! {count, Count}, % 値をメッセージで返す
23       main(Count);
24     s -> % 終了
25       ok;
26     _Other -> % 無視
27       main(Count)
28   end.
29
30 %% 以下、デモ用
31
32 start_demo(Init) ->
```

```
33    % カウンタをスタート
34    start(cntr, Init),
35    % リクエスタをスタート
36    register(req, spawn(fun ()-> req_loop() end)).
37
38  stop_demo() ->
39    cntr ! s,
40    req ! s.
41
42  req_loop() ->
43    receive
44      value ->
45        cntr ! {value, self()}, % カウンタプロセスに中継
46        req_loop();
47      {count, Count} -> % 返答（戻り値）を受け取って、プリント
48        io:format("value = ~p~n", [Count]),
49        req_loop();
50      s -> % 終了
51        ok;
52      _Other -> % 無視
53        req_loop()
54    end.
55
56  %% 以下、インターフェース関数
57
58  up() ->
59    cntr ! up.
60
61  down() ->
62    cntr ! down.
63
64  value() ->
65    req ! value.
```