Parallel Programming Concepts

Actors and Channels

Dr. Peter Tröger
M.Sc. Frank Feinbube

- Part of AI research at MIT
- Another mathematical model for concurrent computation
- No global system state concept (relationship to physics)
- Actor as computational primitive
  - Makes local decisions, has a **mailbox** for incoming messages
  - Concurrently creates more actors
  - Concurrently sends / receives messages
- Asynchronous one-way message sending with changing topology (CSP communication graph is fixed), no order guarantees
  - Recipient is identified by **mailing address**
  - Actors can send their own identity to other actors

„Everything is an actor“
Actor Model

- Interaction with asynchronous, unordered, distributed messaging
- Fundamental aspects
  - Emphasis on local state, time and name space
  - No central entity
  - Actor A gets to know actor B only by direct creation, or by name transmission from another actor C
  - Concurrency utilizes future concept
- Computation
  - Not global state sequence, but partially ordered set of events
    - Event: Receipt of a message by a target actor
    - Each event is a transition from one local state to another
    - Events may happen in parallel
- Messaging reliability declared as orthogonal aspect
Erlang – *Ericsson Language*

- Functional language with actor support, designed for large-scale concurrency
  - First version in 1986 by Joe Armstrong, Ericsson Labs
  - Released as open source since 1998
- Language goals driven by Ericsson product development
  - Scalable distributed execution with large number of concurrent activities
  - Fault-tolerant software under timing constraints
  - Online software update
- Users
  - Amazon EC2 SimpleDB, Delicious, Facebook chat, T-Mobile SMS and authentication, Motorola call processing, Ericsson GPRS and 3G mobile network products, CouchDB, EJabberD, …
Erlang Language

- Sequential subset is influenced by functional and logical programming (Prolog, ML, Haskell, ...)
- Atoms - constant literals, only comparison operation
- Control flow through pattern matching
  ```erlang
  A = 10
  {A, A, B} = {foo, foo, bar}
  ```
- Dynamic typing (runtime even allows invalid types)
- Functions and modules, built-in functions
  - Functions are defined as set of clause
  - On match, all variables in the head become bound.
    ```erlang
    area({square, Side}) -> Side * Side;
    area({circle, rad}) -> math:pi() * rad * rad.
    ```
- Lists and tuples are basis for complex data structures
- Concurrency parts according to the actor model
-module(factorial).
-export([fact/1]).

factorial(0) -> 1;
factorial(N) -> N * factorial(N-1).

> factorial(3).

does not match N = 0 in clause 1
matches N = 3 in clause 2
equal to 3 * factorial(3 - 1)
equal to 3 * factorial(2)
matches N = 2 in clause 2
equal to 3 * 2 * factorial(2 - 1)
equal to 3 * 2 * factorial(1)
matches N = 1 in clause 2
equal to 3 * 2 * 1 * factorial(1 - 1)
equal to 3 * 2 * 1 * factorial(0)
equal to 3 * 2 * 1 * 1 (clause 1)
equal to 6
Sequential Erlang

- **CASE construct:** Result is last expression evaluated on match
  - Catch-all clause not recommended here (‘defensive programming’)  
  - May lead to match error at completely different code position

```erlang
case cond-expression of
  pattern1 -> expr1, expr2, ...
  pattern2 -> expr1, expr2, ...
end
```

- **IF construct:** Test until one of the guards evaluates to TRUE

```erlang
if
  Guard1 -> expr1, expr2, ...
  Guard2 -> expr1, expr2, ...
end
```
Concurrency in Erlang

- **Concurrency Oriented Programming (COP)** [Joe Armstrong]
  - Processes are completely independent (shared nothing)
  - Synchronization and data exchange with message passing
  - Each process has an unforgeable name
  - If you know the name, you can send a message
  - Default approach is fire-and-forget
  - You can monitor remote processes

- Using this gives you
  - No penalty for massive parallelism
  - No huge penalty for distribution (despite latency issues)
  - Fault tolerance basics
  - Concurrency by default
Each concurrent activity is called *process*

Only interaction through *message passing*

Designed for large number of concurrent activities

(Joe Armstrong’s tenets)

□ „The world is concurrent.“
□ „Things in the world don’t share data.“
□ „Things communicate with messages.“
□ „Things fail.“

Design philosophy is to spawn a process for each new event

Constant time to send a message

spawn(module, function, argumentlist)

□ Spawn always succeeds, created process will eventually terminate with a runtime error (‘abnormally‘)
Concurrent Programming in Erlang

-module(tut15).
-export([start/0, ping/2, pong/0]).

ping(0, Pong_PID) ->
    Pong_PID ! finished,
    io:format("ping finished-n", []).

ping(N, Pong_PID) ->
    Pong_PID ! {ping, self()},
    receive
        pong ->
            io:format("Ping received pong-n", [])
    end,
    ping(N - 1, Pong_PID).

pong() ->
    receive
        finished ->
            io:format("Pong finished-n", []).
        {ping, Ping_PID} ->
            io:format("Pong received ping-n", []),
            Ping_PID ! pong,
            pong()
    end.

start() ->
    Pong_PID = spawn(tut15, pong, []),
    spawn(tut15, ping, [3, Pong_PID]).
Counter Actor in Erlang

-module(counter).
-export([run/0, counter/1]).
run() ->
    S = spawn(counter, counter, [0]),
    send_msgs(S, 100000),
    S.
counter(Sum) ->
    receive
        value -> io:fwrite("Value is ~w~n", [Sum]);
        {inc, Amount} -> counter(Sum+Amount)
    end.
send_msgs(_, 0) -> true;
send_msgs(S, Count) ->
    S ! {inc, 1},
    send_msgs(S, Count-1).

% Usage:
% 1> c(counter).
% 2> S = counter:run().
%     ... Wait a bit until all children have run ...
% 3> S ! value.
% Value is 100000
Concurrent Programming in Erlang

- Communication via message passing is part of the language
- Receiver has a **mailbox** concept
  - Queue of received messages
  - Only messages from same source arrive in-order
- Send never fails, works asynchronously (PID ! message)
- Selective message fetching from mailbox
  - receive statement with set of clauses, pattern matching
  - Process is suspended in receive operation until a match

```erlang
receive
    Pattern1 when Guard1 -> expr1, expr2, ..., expr_n;
    Pattern2 when Guard2 -> expr1, expr2, ..., expr_n;
    Other     -> expr1, expr2, ..., expr_n
end
```
Concurrent Programming in Erlang

- Processes can be registered under a name (see shell "regs()".
  - Registered processes are expected to provide a stable service
  - Messages to non-existent processes under alias results in an error on the caller side
- Timeout for receive through additional after block

```erlang
receive
  Pattern1 when Guard1 -> expr1, expr2, ..., expr_n;
  Pattern2 when Guard2 -> expr1, expr2, ..., expr_n;
  Other -> expr1, expr2, ..., expr_n
after
  Timeout -> expr1, expr2, ...
end
```

- Typical process pattern:
  Get spawned, register alias, initialize local state, enter receiver loop with current state, finalize on some stop message
Concurrent Programming in Erlang

- Receiver loop typically modeled with tail-recursive call
  - Receive message, handle it, recursively call yourself
  - Call to sub-routine our yourself is the very last operation, so the stack frame can be overwritten
  - Tail recursion ensures constant memory consumption

- Non-handled messages in the mailbox should be considered as bug, avoid defensive programming (throw away without notice)

- Messaging deadlocks are easily preventable by preventing the \textit{circular wait} condition

- Libraries and templates available for most common patterns
  - Client / Server model - clients access resources and services
  - Finite state machine - perform state changes on message
  - Event handler - receive messages of specific type

- Erlang performs preemptive scheduling (on timeout or receive call)
Example: Tail-Recursion, Read-Only Variables

- For unchanged parameters at the same position, no byte code is generated
- Subroutine call turns into a jump
- No new stack frame per call

```erlang
loop(Module, State) ->
receive
  {call, From, Request} ->
    {Result, State2} = Module:handle_call(Request, State),
    From ! {Module, Result},
    loop(Module, State2);
  {cast, Request} ->
    State2 = Module:handle_cast(Request, State),
    loop(Module, State2)
end.
```
Erlang Robustness

- In massively concurrent systems, you don’t want implicit process dependencies -> Message passing and `spawn` always succeed
- Generic library modules with in-built robustness (e.g. state machines)
- Race conditions are prevented by `selective receive` approach
  - Messages are not processed in order, but based on match only
  - Good for collecting responses for further processing, or rendezvous behavior
  - Transfer of PID supports data sharing with unknown partners

```erlang
PidB!{data, self()}

receive
  {data, PidA}->PidA!response(data)
end
```
Credo:
- „Let it crash and let someone else deal with it“
- „Crash early“

`link()` creates bidirectional link to another process
- If a linked process terminates abnormally, *exit signal* is sent
- On reception, partners send *exit signal* to their partners
  - *Same reason* attribute, leads again to termination

Processes can trap incoming exit signals through configuration, leading to normal message in the inbox

Unidirectional variant `monitor()` for one-way surveillance

Standard build-in atomic function available

```erlang
link(Pid = Spawn(Module, Function, Args))
Pid = spawn_link(Module, Function, Args)
```
Erlang Robustness

- Robustness through layering in process tree
  - Leave processes act as worker (application layer)
  - Interior processes act as supervisor (monitoring layer)
  - Supervisor shall isolate crashed workers from higher system layers through exit trap
  - Rule of thumb: Processes should always be part of a supervision tree
  - Allows killing of processes with updated implementation as a whole -> HA features
Scala - „Scalable Language“

- Martin Odersky, École Polytechnique Fédérale de Lausanne (EPFL)
- Compiler (scalac), Dissassembler (scalap), Console (repl)
- Combination of object oriented and functional language features
  - Expressions, statements, blocks as in Java
  - Every value is an object, every operation is a method call
  - Objects constructed by *mixin-based composition*
  - Functions as first-class concept
- Most language constructs are library functions, can be overloaded
- Compiles to JVM (or .NET) byte code, interacts with class library, re-use of runtime type system
- Example: Twitter moved from Ruby to Scala in 2009

```scala
object HelloWorld extends App {
  println("Hello, world!")
}
```
Scala Basics

- All data types are objects, all operations are methods
- Operator / infix notation
  - 7.5 - 1.5
  - "hello" + "world"
- Object notation
  - (1) . +(2)
  - ("hello") . +("world")
- Implicit conversions, several given by default
  - ("hello") * 5
  - 0. until(3) resp. 0 until 3
  - (1 to 4). foreach (println)
- Type inference
  - var name = "Foo"
- Immutable variables
  - val name = "Scala"
Functions in Scala

- Functions as first-class value - pass as parameter, use as result
- () return value for procedures

```scala
def sum(f: Int => Int, a: Int, b: Int): Int = 
  if (a > b) 0 else f(a) + sum(f, a + 1, b)
def id(x: Int): Int = x
def sumInts(a: Int, b: Int): Int = sum(id, a, b)
def square(x: Int): Int = x * x
def sumSquares(a: Int, b: Int): Int = sum(square, a, b)
```

- Anonymous functions

```scala
def sumSquares(a: Int, b: Int): Int = 
  sum((x: Int) => x * x, a, b)
```
Functions in Scala

- **Parameter type deduction**
  
  ```scala
  def sumSquares(a: Int, b: Int): Int =
  sum((x: Int) => x * x, a, b)
  
  def sumSquares(a: Int, b: Int): Int =
  sum(x => x * x, a, b)
  ```

- **Currying - Transform multiple parameter functions into a chain**
  
  ```scala
  def sum(f: Int => Int, a: Int, b: Int): Int =
  if (a > b) 0 else f(a) + sum(f, a + 1, b)
  
  def sum(f: Int => Int): (Int, Int) => Int = {
    def sumF(a: Int, b: Int): Int =
    if (a > b) 0 else f(a) + sumF(a + 1, b)
    sumF
  }
  
  def sumSquares = sum(x => x * x)
  scala> sumSquares(1, 10)
  ```
Collections in Scala

- Library differentiates between mutable and immutable classes
  - Arrays vs. Lists
  - Two different sub-traits for Set type, differentiation by name space
  - Immutable version of collection as default

```scala
import scala.collection.mutable.Set
val movieSet = Set(“Hitch“, “Poltergeist“)
movieSet += “Shrek“
println(movieSet)
```
Scala Type System
Scala Pattern Matching

- Java `switch` replaced by `match` operation
  - No fall-through semantic
  - At least one match must be given, otherwise `MatchError`
  - Possibility to match for type only (`:`)
  - Possibility to match for an instance (`@`)
  - Default case with `_`

- Pattern matching works everywhere in the code

```scala
val i = ...  
val matched = any match {
  case n: Int => "a number with value: " + n
  case _: String => "a string"
  case true | false => "a boolean"
  case d @ 45.35 => "a double with value " + d
  case d => "an unknown value " + d
}
```
class Rational (n: Int, d: Int) {
    require (d != 0)
    val numer: Int = n
    val denom: Int = d

    override def toString = numer + "/" + denom

def this (n: Int) = this (n, 1)
def *(that: Rational): Rational =
    new Rational(
        numer * that.denom + that.numer * denom,
        denom * that.denom)

def *(i: Int): Rational =
    new Rational (numer*i, denom)
}
Case Classes

- Case classes have
  - Implicit constructor
  - Accessor methods for constructor arguments
  - Implementations of `toString`, `equals`, `hashCode`

- Two case class members are equal if they had the same construction parameters, so this yields 'True':

  ```scala
  Sum(Number(1), Number(2)) == Sum(Number(1), Number(2))
  ```

- Foundation for pattern matching

```scala
def eval(e: Expr): Int = e match {
  case Number(n) => n       // matches all Number(v) values
  case Sum(l, r) => eval(l) + eval(r)
}
Execution as Substitution

- $\text{eval}(\text{Sum}(\text{Number}(1), \text{Number}(2)))$

- $\text{Sum}(\text{Number}(1), \text{Number}(2))$ match {
  case $\text{Number}(n)$ => $n$
  case $\text{Sum}(e1, e2)$ => $\text{eval}(n1) + \text{eval}(n2)$
}

- $\text{eval}(\text{Number}(1)) + \text{eval}(\text{Number}(2))$

- $\text{Number}(1)$ match {
  case $\text{Number}(n)$ => $n$
  case $\text{Sum}(e1, e2)$ => $\text{eval}(n1) + \text{eval}(n2)$
} + $\text{eval}(\text{Number}(2))$

- $1 + \text{eval}(\text{Number}(2))$

- $1+2$

- $3$
Example: Quicksort

```scala
def sort(xs: Array[Int]) {
  def swap(i: Int, j: Int) {
    val t = xs(i)
    xs(i) = xs(j); xs(j) = t; ()
  }
  def sort1(l: Int, r: Int) {
    val pivot = xs((l + r) / 2)
    var i = l; var j = r
    while (i <= j) {
      while (xs(i) < pivot) i += 1
      while (xs(j) > pivot) j -= 1
      if (i <= j) {
        swap(i, j); i += 1; j -= 1
      }
    }
    if (l < j) sort1(l, j)
    if (i < r) sort1(i, r)
  }
  sort1(0, xs.length - 1)
}
```

- Similar to imperative languages
- Functions in functions, global variables
- Read-only value definition with `val`
Example: Quicksort

- Functional style (same complexity, higher memory consumption)
  - Return empty / single element array as already sorted
  - Partition array elements according to pivot element
  - Higher-order function `filter` takes predicate function ("pivot > x") as argument and applies it for filtering
  - Sorting of sub-arrays with predefined `sort` function

```scala
def sort(xs: Array[Int]): Array[Int] = {
  if (xs.length <= 1)
    xs
  else {
    val pivot = xs(xs.length / 2)
    Array.concat(
      sort(xs filter (pivot >)),
      xs filter (pivot ==),
      sort(xs filter (pivot <)))
  }
}
```
Concurrent Programming with Scala

- Actor-based concurrent programming, similar to Erlang
  - Concurrency abstraction on-top-of threads or processes
  - Communication by asynchronous send operation and synchronous receive block
    ```scala
    actor {
      var sum = 0
      loop {
        receive {
          case Data(bytes)       => sum += hash(bytes)
          case GetSum(requester) => requester ! sum
        }
      }
    }
    ```
- All constructs are library functions (actor, loop, receiver, !)
- Alternative `self.receiveWithin()` call with timeout
- Case classes act as message type representation
- With Scala 2.11, actor implementation will come from AKKA library
import scala.actors.Actor
import scala.actors.Actor._
case class Inc(amount: Int)
case class Value

class Counter extends Actor {
  var counter: Int = 0;
  def act() = {
    while (true) {
      receive {
        case Inc(amount) =>
          counter += amount
        case Value =>
          println("Value is "+counter)
          exit()
      }
    }
  }
}

object ActorTest extends Application {
  val counter = new Counter
  counter.start()
  for (i <- 0 until 100000) {
    counter ! Inc(1)
  }
  counter ! Value
  // Output: Value is 100000
}
AKKA Actors

- Actor has a **receive** method, which returns a partial function
  - Calls the function with the incoming message
- Each actor instance has it's **mailbox**
  - If the actor is not running in another execution context, it is allocated to one thread and called with the message
- All actors are part of the **ActorSystem**, must be used for creation
  - Returns **ActorRef** that can be serialized

```scala
sealed trait Request
case object ARequest extends Request
case object BRequest extends Request

import akka.actor.Actor
class Server extends Actor {
  def receive = {
    case ARequest => println("Request type A")
    case BRequest => println("Request type B")
  }
}
```
Concurrent Programming with Scala

- Sending messages
  - tell (or !)
    - Sends a message asynchronously and returns immediately
  - Ask (or ?)
    - Sends a message asynchronously and returns a Future

- In-built support for Finite State Machine (FSM) actors (AKKA)

- Property concept to influence the execution strategy (AKKA)

- Support for parallel collections (since 2.9)

- Software transactional memory is available through libraries

```scala
Case class Increment(amount: Int)

Class Counter extends Actor {
  private var count = 0
  def receive = {
    case Increment(by) =>
      count += by
      println(count)
  }
}
```
Case class Increment(amount: Int)

Class Counter extends Actor {
  private var count = 0
  def receive = {
    case Increment(by) =>
      count += by
      println(count)
  }
}

Val counter = actorOf[Counter]  // use actor system
counter.start
counter ! Increment(10)
futureVar = counter !!! Increment(10)
Implicit superclass is `scala.AnyRef`, provides typical monitor functions

```scala
scala> classOf[AnyRef].getMethods.foreach(println)
def wait()
def wait(msec: Long)
def notify()
def notifyAll()
```

Synchronized function, argument expression is executed exclusive
```
def synchronized[A] (e: => A): A
```

Synchronized variable with `put`, blocking `get` and `unset`
```
val v=new scala.concurrent.SyncVar()
```

Futures, reader / writer locks, semaphores, mailboxes, ...
```
import scala.concurrent.ops._
...
val x = future(someLengthyComputation)
anotherLengthyComputation
val y = f(x()) + g(x())
```

Explicit parallelism through `spawn (expr)`
Case Classes are Message Types

```scala
import scala.actors.Actor

abstract class AuctionMessage

case class Offer(bid: Int, client: Actor) extends AuctionMessage

case class Inquire(client: Actor) extends AuctionMessage

abstract class AuctionReply

case class Status(asked: Int, expire: Date) extends AuctionReply

case object BestOffer extends AuctionReply

case class BeatenOffer(maxBid: Int) extends AuctionReply

case class AuctionConcluded(seller: Actor, client: Actor) extends AuctionReply

case object AuctionFailed extends AuctionReply

case object AuctionOver extends AuctionReply
```
class **Auction**(seller: Actor, minBid: Int, closing: Date) extends Actor {

    val timeToShutdown = 36000000  // inform that auction was closed
    val bidIncrement = 10

    def act() {
        var maxBid = minBid - bidIncrement;  var maxBidder: Actor = null;  var running = true
        while (running) {
            receiveWithin ((closing.getTime() - new Date().getTime())) {
                case Offer(bid, client) => if (bid >= maxBid + bidIncrement) {
                    if (maxBid >= minBid) maxBidder ! BeatenOffer(bid)
                    maxBid = bid;  maxBidder = client;  client ! BestOffer }
                else client ! BeatenOffer(maxBid)
                case Inquire(client) => client ! Status(maxBid, closing)
                case TIMEOUT => if (maxBid >= minBid) {
                    val reply = AuctionConcluded(seller, maxBidder)
                    maxBidder ! reply;  seller ! reply }
                else seller ! AuctionFailed receiveWithin(timeToShutdown) {

...
Alternative *react* function, also takes partial function as input for the decision, but does not return on match

- Another tail recursion case – implementable by one thread
- Message handler must process the message and do all work
- Typical idiom is to have top-level work method being called

```scala
object NameResolver extends Actor {
  import java.net.InetAddress
  def act() {
    react {
      case (name: String, actor: Actor) =>
        actor ! InetAddress.getByName(name)
        act()
      case "EXIT" =>
        println("Exiting")
      case msg =>
        println("Unknown message")
        act()
    }
  }
}
```
Actor Deadlocks

- Synchronous send operator „!?” available in Scala

```scala
Actor deadlock: Actor A
1 actorB !? Msg1(value) match {
2   case Response1(r) =>
3     // ...
4 }
5
6 receive {
7   case Msg2(value) =>
8     reply(Response2(value))
9 }

Actor deadlock: Actor B
1 actorA !? Msg2(value) match {
2   case Response2(r) =>
3     // ...
4 }
5
6 receive {
7   case Msg1(value) =>
8     reply(Response1(value))
9 }
```

```scala
Safe loop: Actor A
1 actorB ! Msg1(value)
2 while (true) {
3   receive {
4     case Msg2(value) =>
5       reply(Response2(value))
6     case Response1(r) =>
7       // ...
8   }
9 }

Safe loop: Actor B
1 actorA ! Msg2(value)
2 while (true) {
3   receive {
4     case Msg1(value) =>
5       reply(Response1(value))
6     case Response2(r) =>
7       // ...
8   }
9 }
```

PROC producer (CHAN INT out!)
    INT x:
    SEQ
        x := 0
        WHILE TRUE
            SEQ
                out ! x
                x := x + 1
     :

PROC consumer (CHAN INT in?)
    WHILE TRUE
        INT v:
        SEQ
            in ? v
            .. do something with `v'
     :

PROC network ()
    CHAN INT c:
    PAR
        producer (c!)
        consumer (c?)
     :
actor {
  var out: OutputChannel[String] = null
  val child = actor {
    react {
      case "go" => out ! "hello"
    }
  }
  val channel = new Channel[String]
  out = channel
  child ! "go"
  channel.receive {
    case msg => println(msg.length)
  }
}

Sending channels in messages

```scala
case class ReplyTo(out: OutputChannel[String])

val child = actor {
  react {
    case ReplyTo(out) => out ! "hello"
  }
}

actor {
  val channel = new Channel[String]
  child ! ReplyTo(channel)
  channel.receive {
    case msg => println(msg.length)
  }
}
```

Scope-based channel sharing
Go

- Programming language developed by Google since 2007
- Written by Robert Griesemer, Rob Pike, and Ken Thompson
- Compiler for various platforms (X86, X64, ARM)
- UTF-8 everywhere, immutable strings
- Only control statements are `if`, `for`, `switch`, and `defer`
- No exceptions by design, only error codes
- Automatic garbage collection
- „Goroutine“
  - Executed concurrently with other goroutines
  - Not to be confused with threads, processes, or coroutines
  - Declaration by function prefix „go“
  - Scheduler switches to another routine on blocking
- Currently not intended for parallel code, concurrent language
  - [http://blog.golang.org/concurrency-is-not-parallelism](http://blog.golang.org/concurrency-is-not-parallelism)
Goroutines

- Same address space for all routines
  - Compiler multiplexes to some thread(s) - configurable
  - Language and compilers are designed for extrem concurrency (thousands to millions goroutine instances)

- Communication through channels
  - ch1 := make(chan int) // create channel
  - v = <-c       // receive from c, assign to v
  - <-c          // receive and throw away
  - var recvChan <-chan int // Receive only channel
  - var sendChan chan<- int // Send only channel

- Channel communication blocks until the matching receive
  - Practical realization of Hoare\'s CSP in a language (!)

- make can also create buffered channels for asychronous send
package main
import "fmt"

func sayHello (ch1 chan string){
    ch1<-"Hello World\n"
}

func main() {
    ch1 := make(chan string)
    go sayHello(ch1)
    fmt.Printf(<-ch1)
}

$ 8g chanHello.go ; 8l -o chanHello chanHello.8
$ ./chanHello
Hello World
Go Channels

- **select** statement allows to switch between available channels
  - All channels are evaluated
  - If multiple can proceed, one is chosen randomly
  - Default clause if no channel is available

```go
select {
    case v := <-ch1:
        fmt.Println("channel 1 sends", v)
    case v := <-ch2:
        fmt.Println("channel 2 sends", v)
    default: // optional
        fmt.Println("neither channel was ready")
}
```

- Channels are first-class language values, can be returned / passed
  - Example: Client provides a response channel in the request
- Type-safe distributed channels with „`netchan`` package
Example: The `timeout` channel

```go
func main() {
    c := boring("Joe")
    for {
        select {
            case s := <-c:
                fmt.Println(s)
            case <-time.After(1 * time.Second):
                fmt.Println("You're too slow.")
                return
        }
    }
}
```