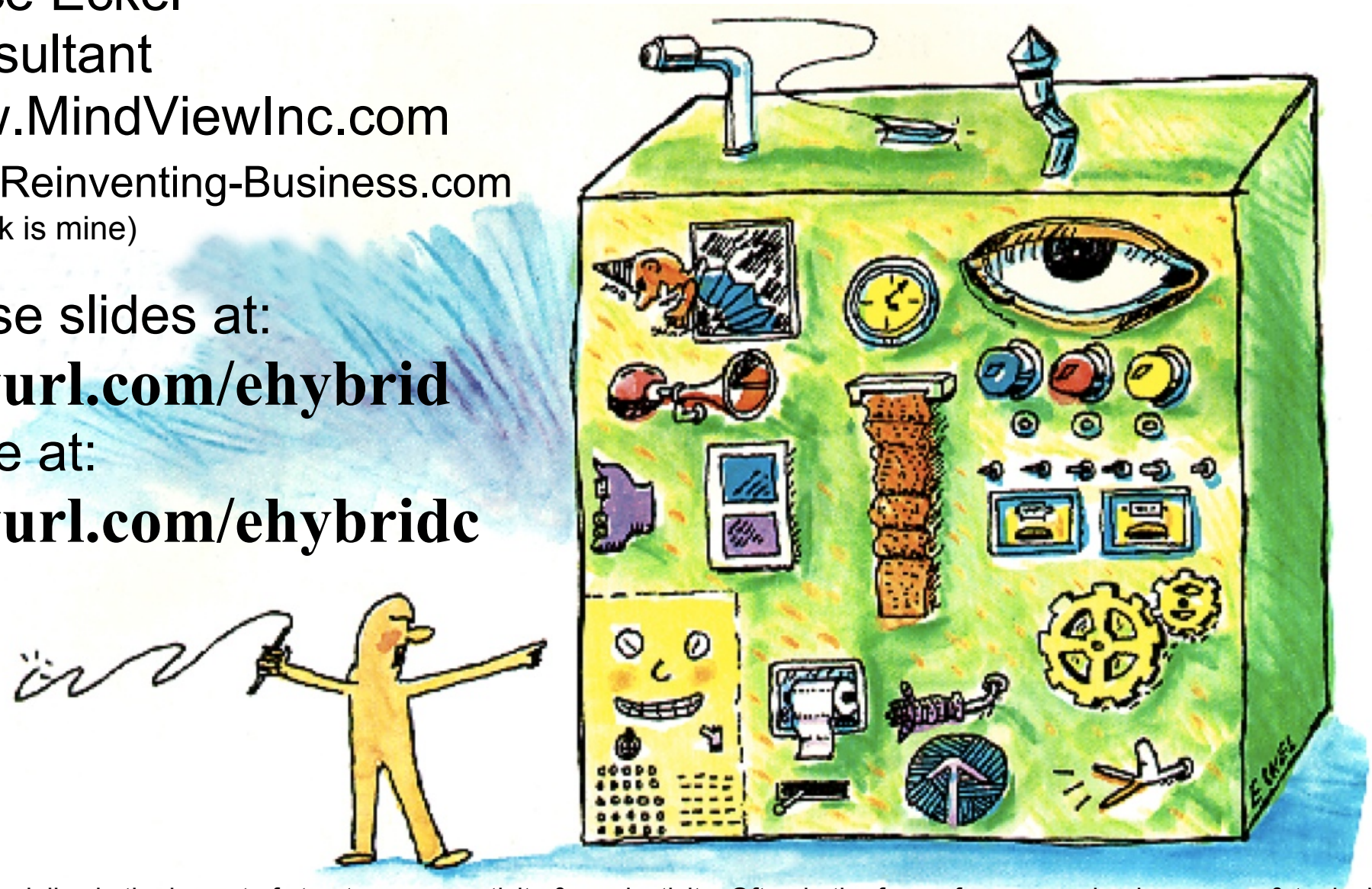
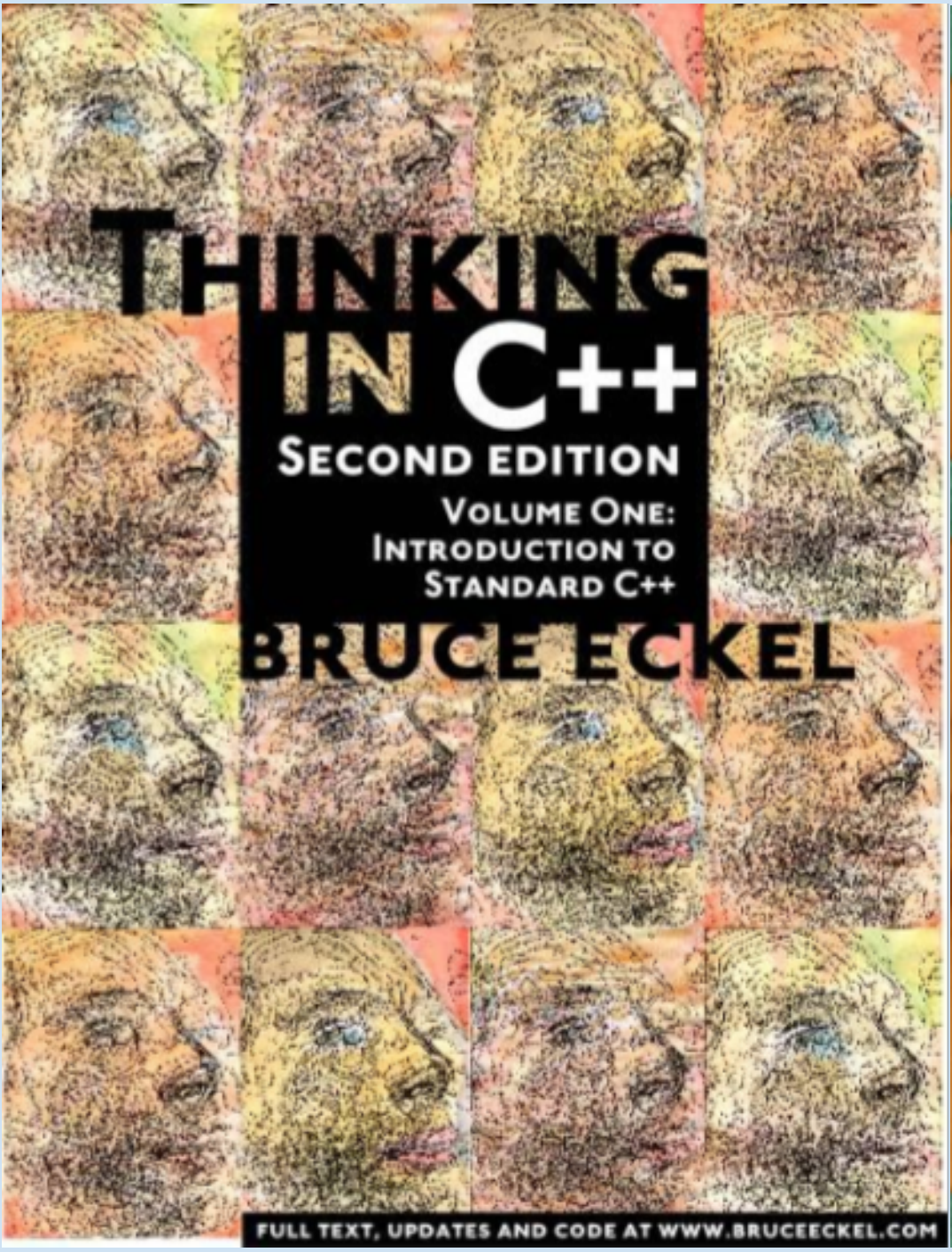
The Power of Hybridization

Bruce Eckel
Consultant
www.MindViewInc.com
www.Reinventing-Business.com
(Artwork is mine)

These slides at:
tinyurl.com/ehybrid
Code at:
tinyurl.com/ehybridc



- I specialize in the impact of structure on creativity & productivity. Often in the form of programming languages & tools, but not limited.
- Changes since I did the abstract
- All the research changed much of my thinking.


THINKING IN C++
SECOND EDITION
VOLUME ONE: INTRODUCTION TO STANDARD C++
BRUCE ECKEL
FULL TEXT, UPDATES AND CODE AT WWW.BRUCEECKEL.COM


BRUCE ECKEL
THINKING IN JAVA
4TH EDITION
The definitive introduction to object-oriented programming in the language of the world wide web
FOR JAVA SE5/6
Software Development Jolt Award & Productivity Award
Java World Editor's Choice, Reader's Choice
Java Developer's Journal Editor's Choice, Reader's Choice
CODE & SUPPLEMENTS AT WWW.MINDVIEW.NET

Learn Flex Fast

This book will take you through your first steps on your way to becoming a powerful user interface programmer.

You'll learn everything you need to know to begin building applications and exploring the Flex landscape – but only what you need to know.

We've gone to great lengths to show you the world of Flex without burying you in information you don't need right now. At the same time, we give pointers to places where you can go to explore more.

First Steps in Flex is the ideal starting point for any programmer who wants to quickly become proficient in Flex 3.

Bruce Eckel (www.MindViewInc.com) specializes in languages and rapid-development tools and techniques, and provides consulting, workshops and conferences. He is the author of the multi-award-winning Thinking in Java and Thinking in C++, among others.

James Ward (www.JamesWard.org) is a Technical Evangelist for Flex at Adobe. He travels the globe speaking at conferences and teaching developers how to build better software with Flex.

First Steps in Flex

ISBN 978-0-9818725-0-6
52400
9 780981 872506
Price: $24.00

cover design by will-harris.com

FIRST STEPS IN FLEX By Bruce Eckel & James Ward

By Bruce Eckel
President, MindView Inc.
& James Ward
Flex Evangelist, Adobe Inc.

First Steps in Flex

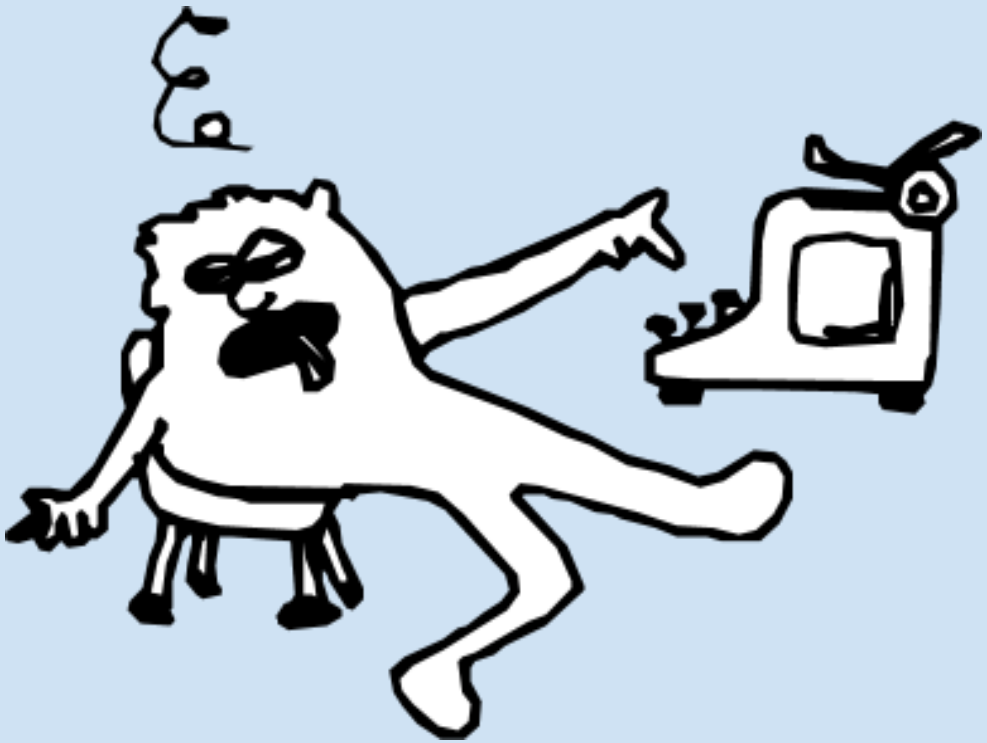
Building Applications with Flex 3

I Don't Do Realism

- (For Context)
- Abstract paintings
- Cartoons
- Software
- Crested Butte
- Open-Spaces style learning events
- Theatre
- Burning Man
- Never satisfied with current languages/tools
- Organizational structure
- Feel free to dismiss all this as too idealistic

Why?

- Learning to program & learning to write: not *that* hard
- Becoming a good programmer or writer: life work

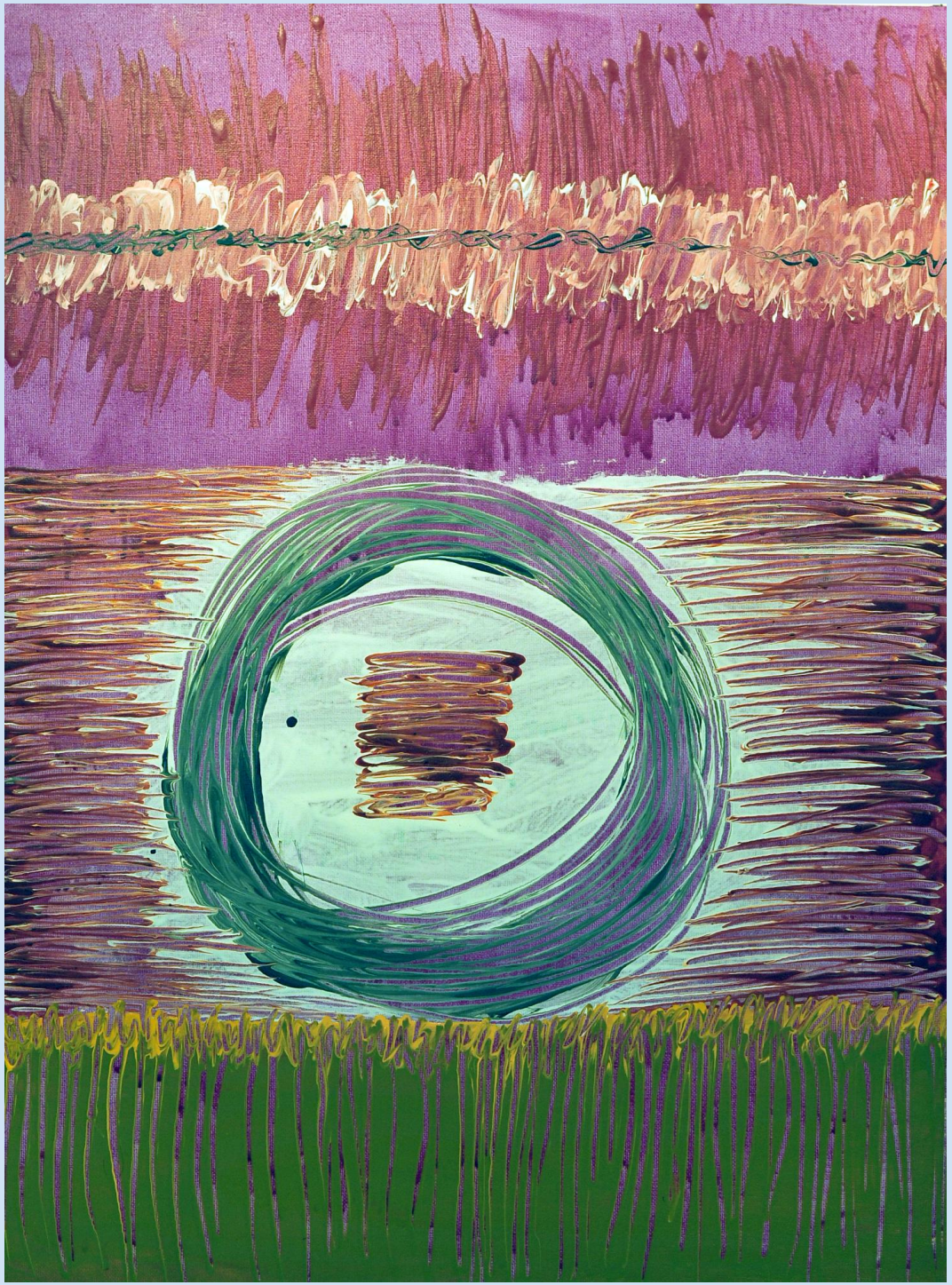


As a Programmer

You can learn to be useful with your tools. This is good.

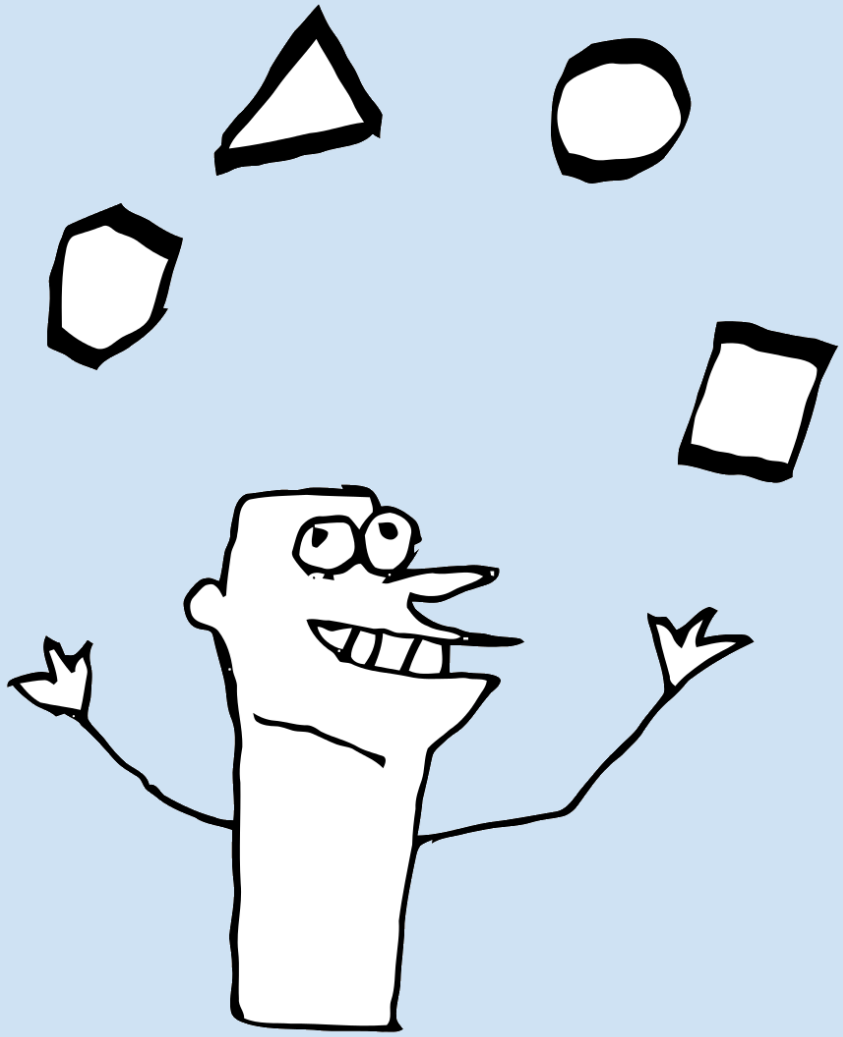
OR

You can always be searching for ways to cheat.
I think this is better.



Yes, you're a good programmer

But, just this once, let's assume that the problem you're tackling, no matter how simple it seems, will end up being overwhelming.



Let's start out thinking you need to take every advantage and shortcut necessary to keep the solution as simple as possible.

Remember *cognitive load.*

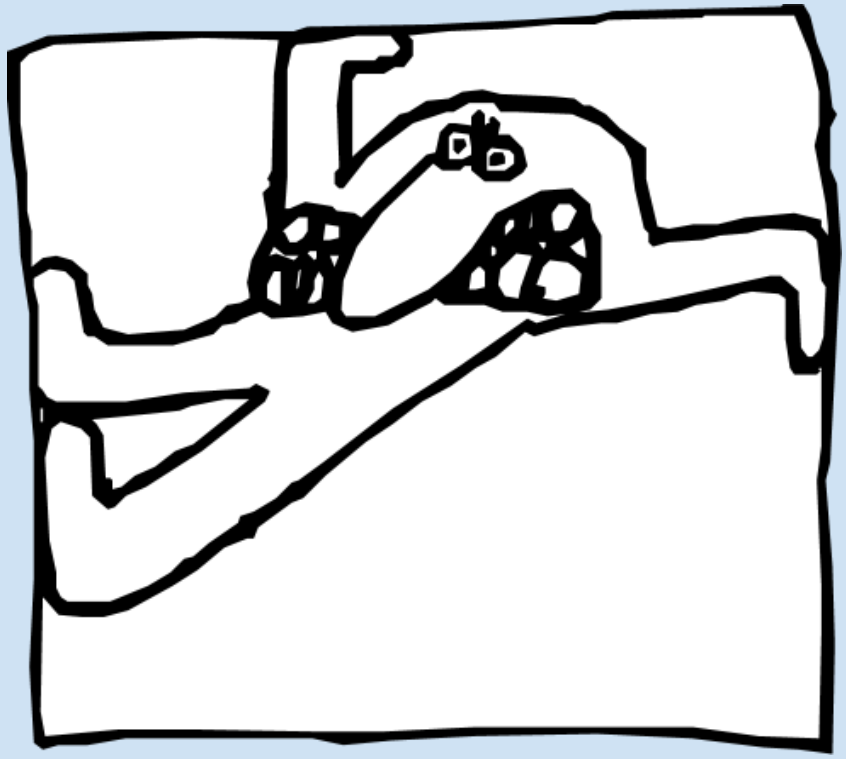
Perhaps languages should no longer be sacred cows.

And...

Let's assume, just this once, that you don't know ahead of time where all the difficult parts will be, and so you'd like to race through and get something up and running so you can discover where those things are.

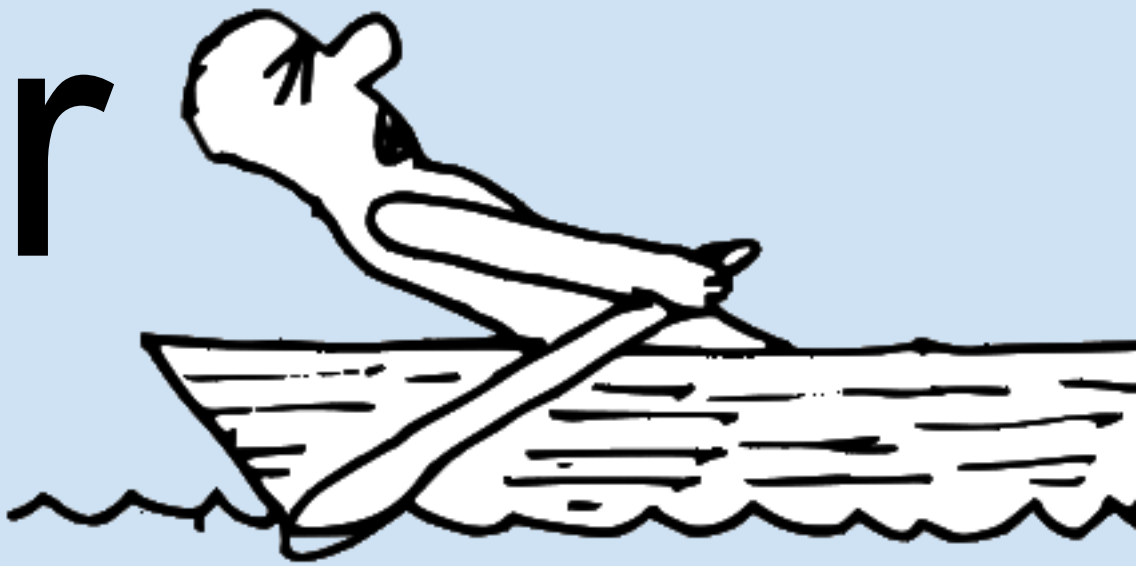
You need every shortcut you can find.

Do the Simplest Thing



Fight Premature Optimization

Assume it will be (a lot) Harder



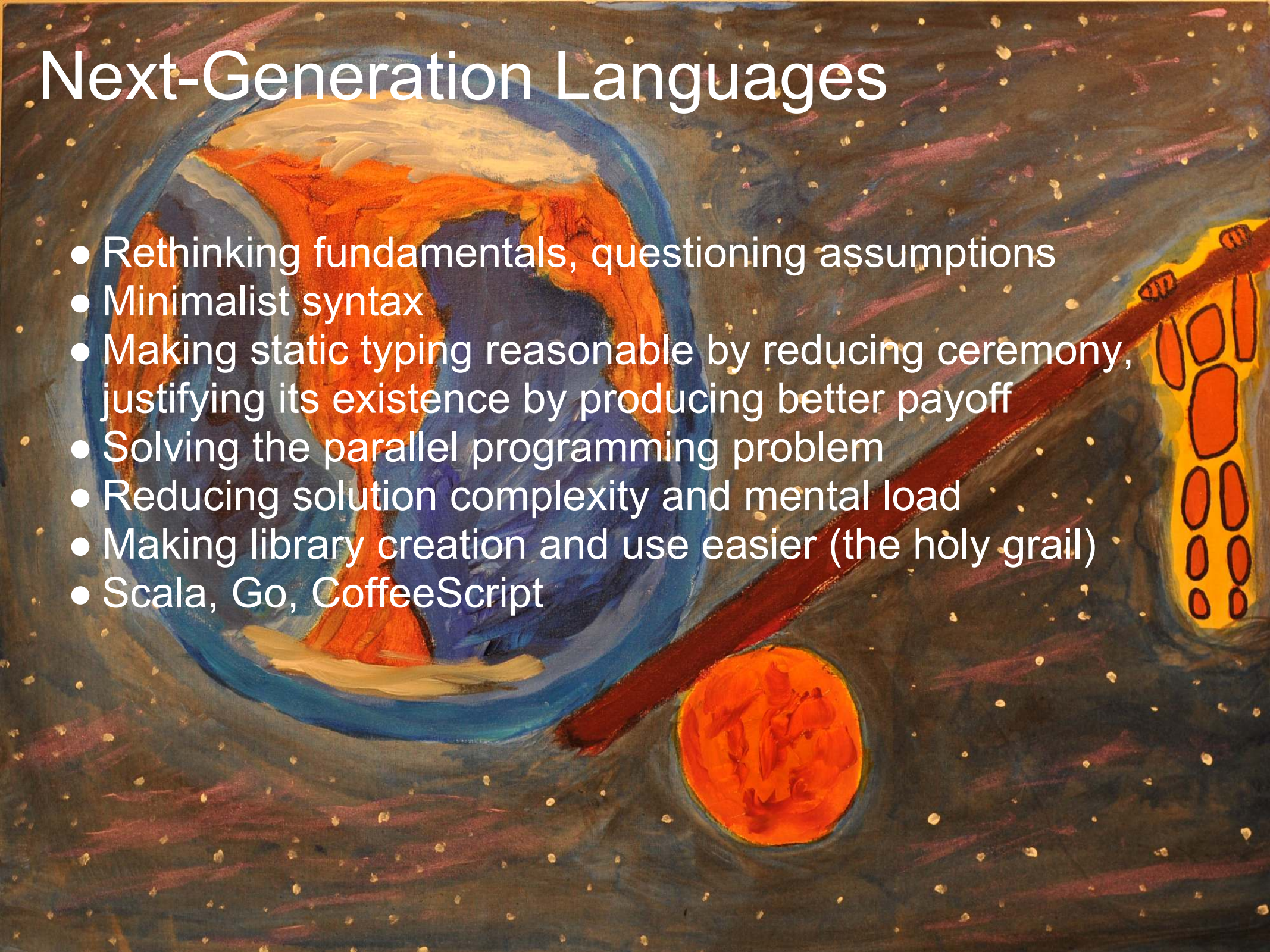
Stack the Deck

Dick Wall Mongo DB solution

"Mainstream" Language Timeline

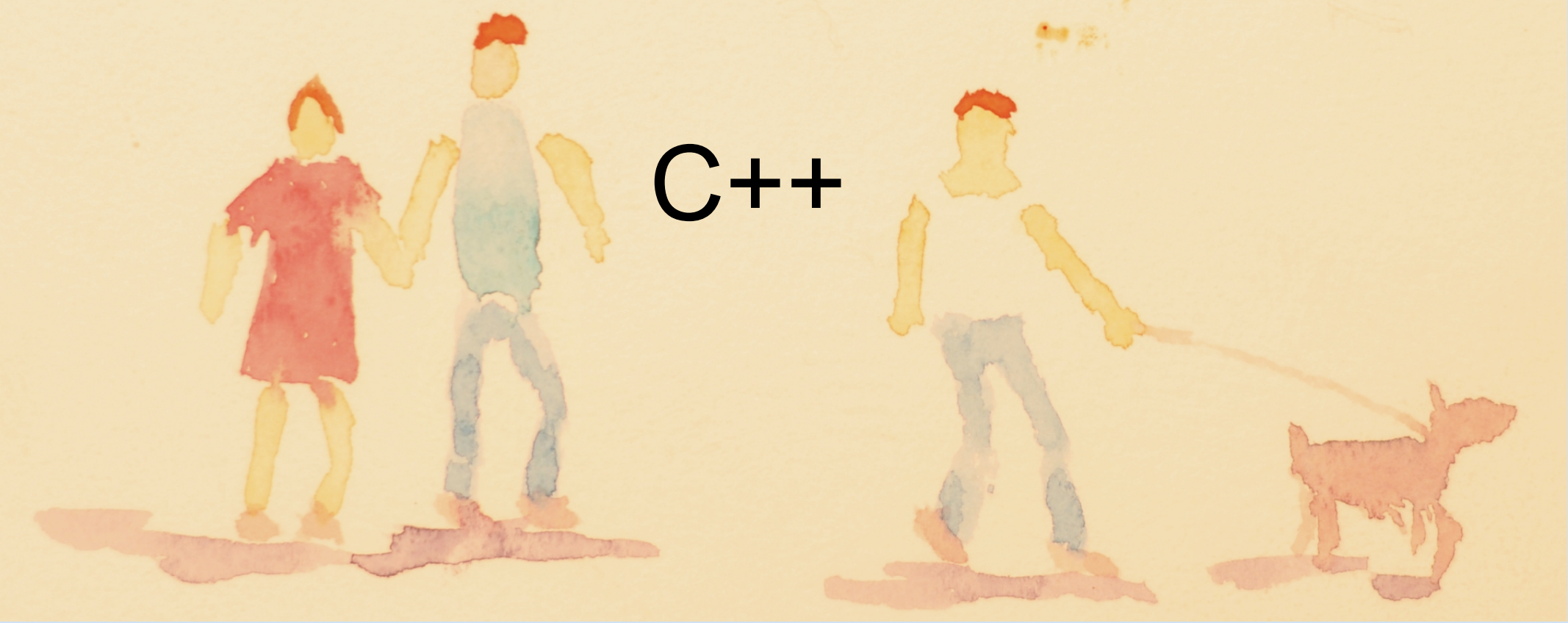
- (I learned BASIC first)
- Assembly + procedural, Pascal & C
- When that got out of control: "Structured Revolution"
- C++ dragged us into OO
- Booch, Rumbaugh, Jacobsen battles resolved with UML
- Java dragged us into Garbage Collection, Unified Error Handling, Virtual Machines
- Agile Manifesto resolves battles between anti-waterfall camps (rapid design/implementation cycles, high stakeholder communication) (waterfall power of story)
- Dynamic languages a reaction to excessive ceremony/hoop-jumping by C++ and Java

Next-Generation Languages

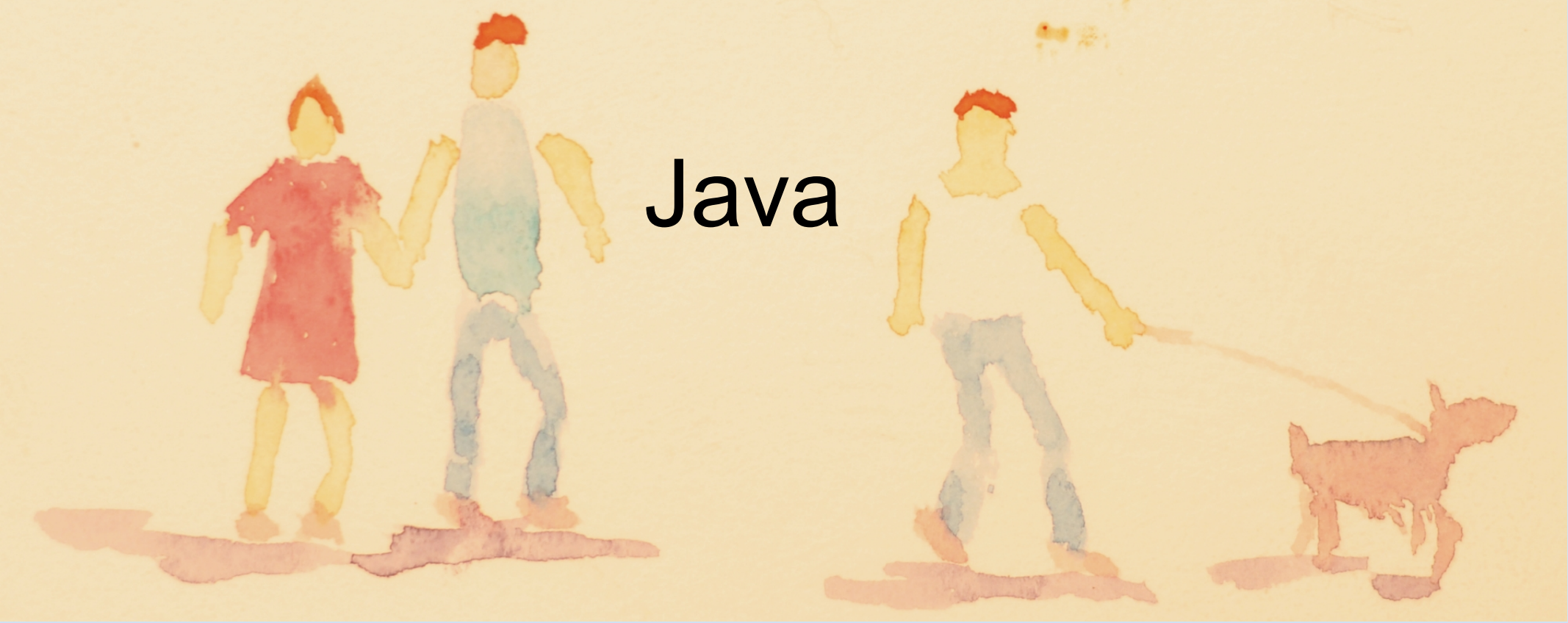


- Rethinking fundamentals, questioning assumptions
- Minimalist syntax
- Making static typing reasonable by reducing ceremony, justifying its existence by producing better payoff
- Solving the parallel programming problem
- Reducing solution complexity and mental load
- Making library creation and use easier (the holy grail)
- Scala, Go, CoffeeScript

All Languages Fail at Some Things

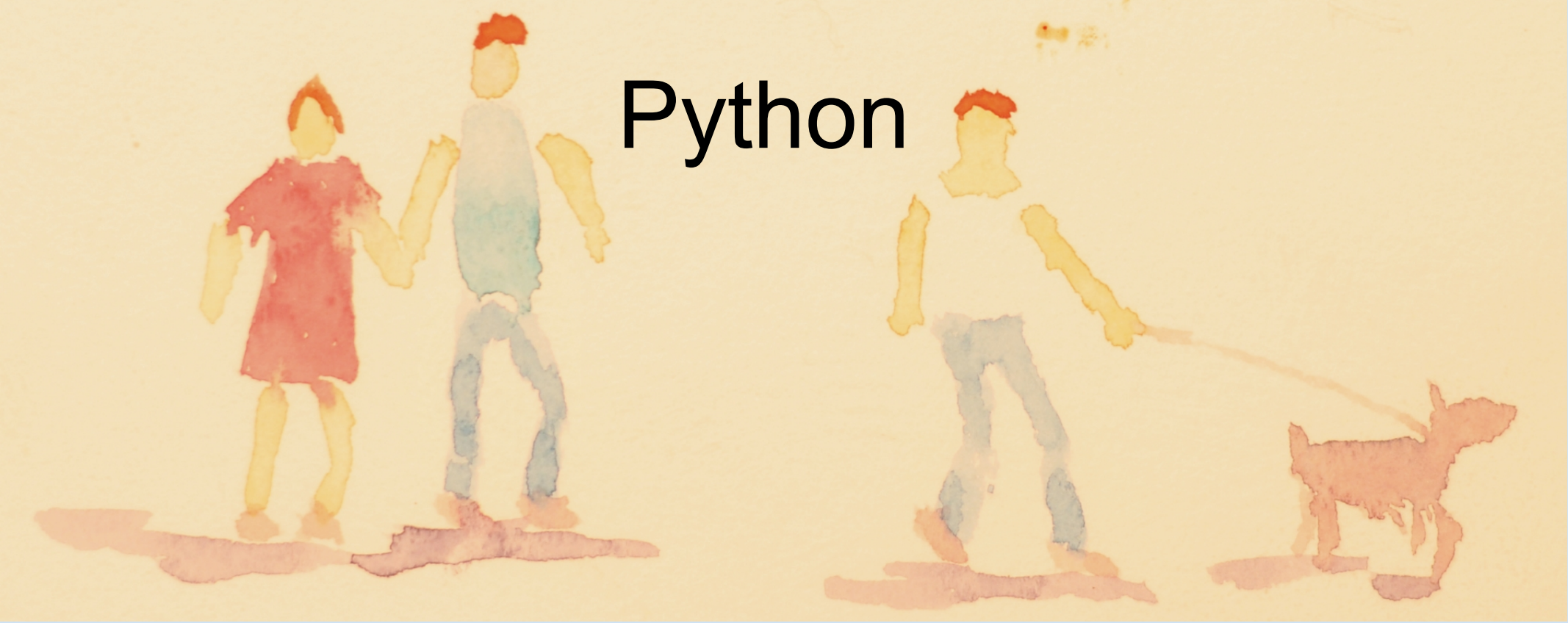


- Too much work for backwards compatibility with C
- No GC, no concurrency or parallelism (hooks in C++0X)
- No UI (Qt seems dominant?)

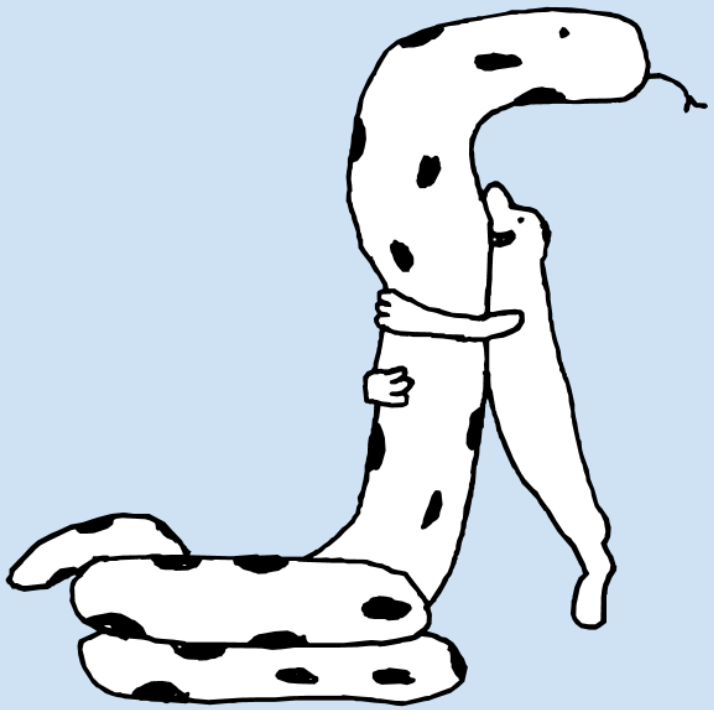


Java

- Too much impact for backwards compatibility with itself (e.g.: Programmer must know all about generics just to *use* them)
- Reams of ceremony/boilerplate produces hard-to-understand systems
- Bad parallelism (shared memory)
- Multiple failed attempts at UI (The downside of commercial-driven language design)



Python



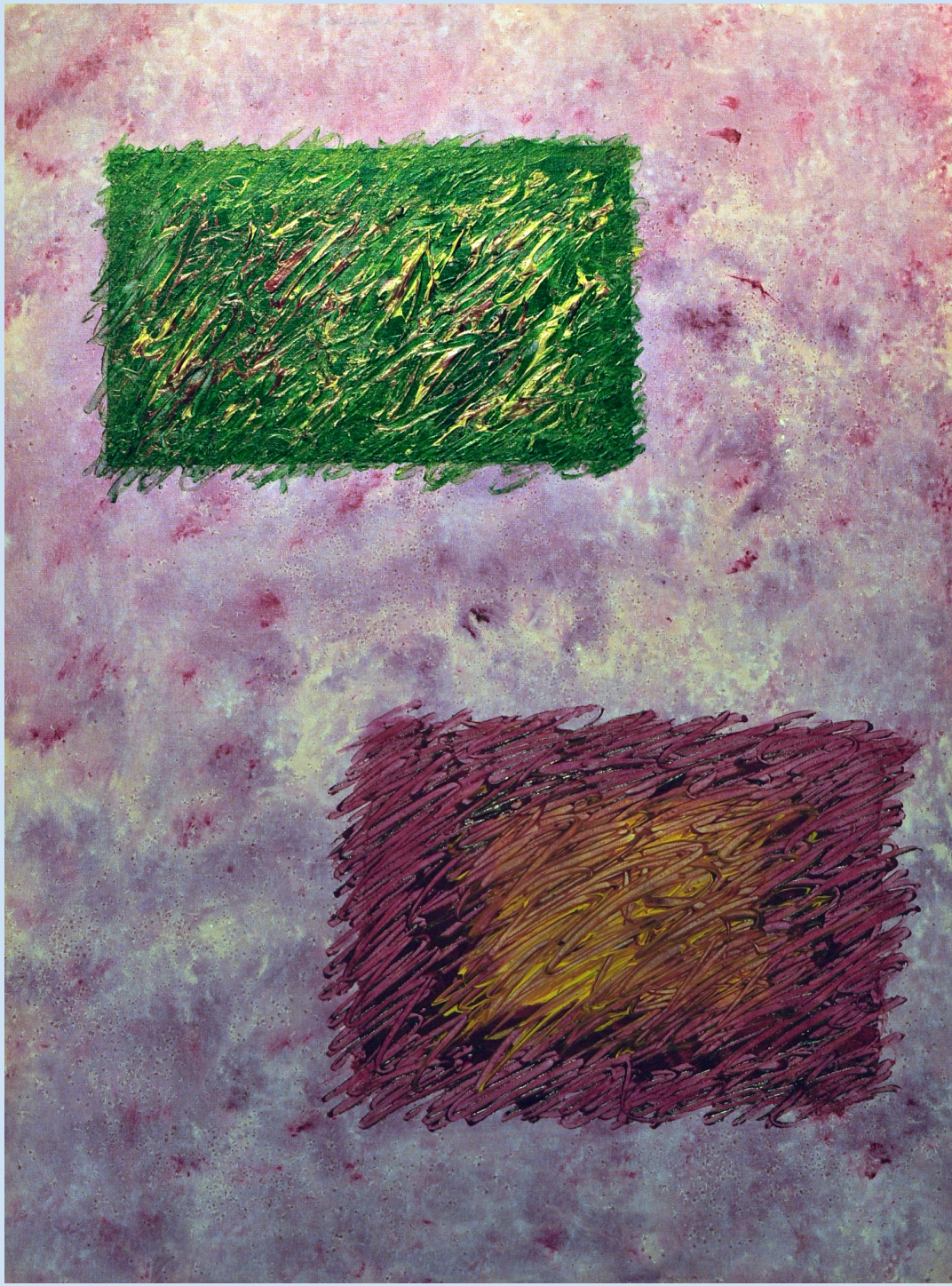
- No thread-parallelism (GIL)
- (But **multiprocessing** for process-parallelism, as you'll see)
- UI libraries: nothing easy, all suffer from corner cases (but Qt @ ALBA)
- Speed can be an issue (but not as often as you think)

Example: Concurrency & Parallelism

- Concurrency vs. Parallelism
  - Concurrency is a *design approach*: decompose into "independently executing" pieces
  - Concurrency enables parallelism
- Shared memory vs message-passing
  - This affects both concurrency & parallelism
- Just because threads (shared-memory concurrency) seem to make sense doesn't mean you can program with threads (you can't)
- You can't test for correctness (you can only discover failure)

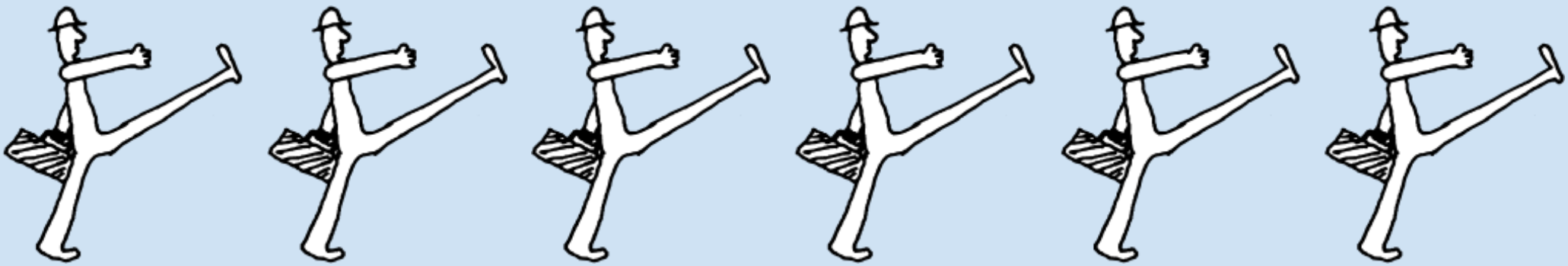
Solution

- Don't make me think
- CSP: *Communicating Sequential Processes*
- Completely isolated processes, with guaranteed safe communication
- Proof of concept: novices can safely program (definitely *not* true for threads)



multiprocessing: more Python brilliance

- Python and Ruby have threading modules, but cannot utilize more than one CPU because of the Global Interpreter Lock (GIL)
- Solution: Use OS processes. I built a system for one client several years ago that is now being used in 10 models
- Since then, Python has incorporated the **multiprocessing** module (in 2.6), producing both local and remote concurrency. Has synchronized queues/pipes.
- Other languages might be faster, but with enough CPUs it's often fast enough -- and faster-to-create, producing more manageable code.



```
# ParallelDemo.py
import random, time, multiprocessing

def long_running():
    delay = random.randrange(2, 12)
    print("Starting: %d" % delay)
    time.sleep(delay)
    print("Finished: %d" % delay)

for n in range(10):
    multiprocessing.Process(
        target=long_running).start()
```

```
# ParallelDemo.py
import random, time, multiprocessing

def long_running():
    delay = random.randrange(2, 12)
    print("Starting: %d" % delay)
    time.sleep(delay)
    print("Finished: %d" % delay)

for n in range(10):
    multiprocessing.Process(
        target=long_running).start()
```

"But I Have Too Much Java Code!"

- Are you relegated to struggling with Java threads?
  (Even with the improvements, it's tricky)
- No: Scala works transparently with existing Java code.
- Has Actors!
- My article introducing Scala
- I'm coauthoring a simple introduction to Scala

Java:

```
public class MyClass {
  public double myMethod(double d) {
    return d * 7;
  }
}
```

Scala script transparently calls Java:

```
val rand = new java.util.Random
val m = new MyClass
println(m.myMethod(rand.nextDouble))
```

Java:

```
public class MyClass {
  public double myMethod(double d) {
    return d * 7;
  }
}
```

Scala script transparently calls Java:

```
val rand = new java.util.Random
val m = new MyClass
println(m.myMethod(rand.nextDouble))
```

Java:

```
public class MyClass {
  public double myMethod(double d) {
    return d * 7;
  }
}
```

Scala script transparently calls Java:

```
val rand = new java.util.Random
val m = new MyClass
println(m.myMethod(rand.nextDouble))
```

Java:

```
public class MyClass {
  public double myMethod(double d) {
    return d * 7;
  }
}
```

Scala script transparently calls Java:

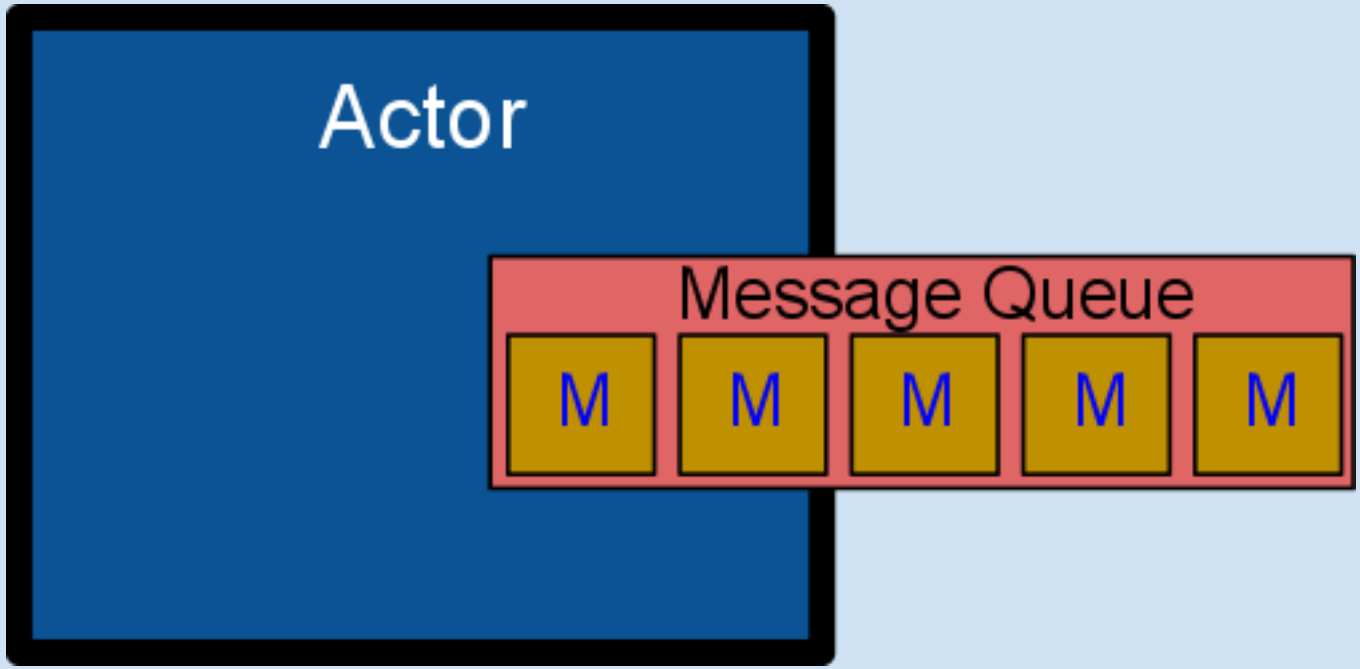
```
val rand = new java.util.Random
val m = new MyClass
println(m.myMethod(rand.nextDouble))
```

Java:

```
public class MyClass {
  public double myMethod(double d) {
    return d * 7;
  }
}
```

Scala script transparently calls Java:

```
val rand = new java.util.Random
val m = new MyClass
println(m.myMethod(rand.nextDouble))
```



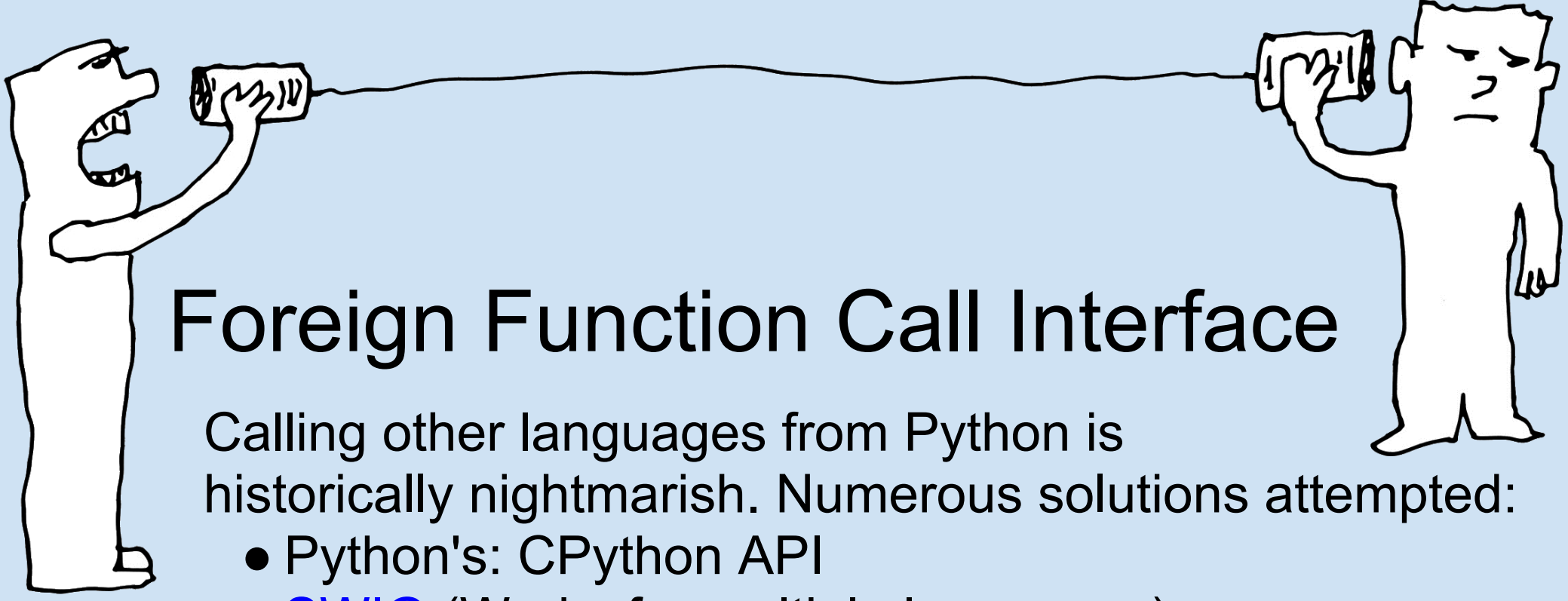
- **myActor ! myMessageObject**
- Can select any message in queue, not just first one
- Just one way to do CSP
- Go uses goroutines and channels:
  go doStuff(chan)
  chan <- myMessageObject
- Python **multiprocessing** uses **Queue**s (one-way) and **Pipe**s (two-way)

Scala Parallel Collections

```
val result =
    toBeProcessed.par.map(obj => process(obj))
```

Akka library

- Uses Actors and Software Transactional Memory so you can write simpler, correct concurrent apps
- Scale across cores or nodes using asynchronous message passing (can even use GPU for floating calculations!)
- Transparent remoting with Actors
- Open-source, will eventually be part of standard Scala
- Also works with Java



Foreign Function Call Interface

Calling other languages from Python is historically nightmarish. Numerous solutions attempted:

- Python's: CPython API
- SWIG (Works for multiple languages)
- Cython C-extensions
- Boost.Python for C++ Interoperability
- Pyrex: Write simple C-like code for extensions
- Python's **ctypes** library dynamically binds very efficiently to DLLs or shared libraries. Can still be tricky.
- SIP
- Shiboken (for C++; genesis)

Consider Remote Procedure Calls

- Uses an existing transport mechanism to make calls (sockets)
- Pass/return data structures in prescribed format and layout
- Tends to be the simplest approach
- Some overhead for call, but remote operation should be the heavier cost, not the call (tune this)
- XML-RPC was popular and has Python standard library support
- JSON-RPC more recent, supported by Go, Python libs available

Go Language

- Modern: minimized syntax
- Open Source from Google, very active, version 1 soon.
- Rob Pike, Ken Thompson full time team leaders
- Designed to create servers, becoming general-purpose systems language (C++ replacement)
- Core support for "goroutines" and synchronized pipes
- Library support for JSON-RPC
- My articles calling from Python to Go via JSON-RPC

```
package main
import (
  "rpc/jsonrpc"
  "rpc"
  "os"
  "net"
  "log"
)

type RPCFunc uint8

func (*RPCFunc)
Echo(arg *string, result *string) os.Error {
  log.Print("Arg passed: " + *arg)
  *result = ">" + *arg + "<"
  return nil
}
```

```
package main
import (
  "rpc/jsonrpc"
  "rpc"
  "os"
  "net"
  "log"
)

type RPCFunc uint8

func (*RPCFunc)
Echo(arg *string, result *string) os.Error {
  log.Print("Arg passed: " + *arg)
  *result = ">" + *arg + "<"
  return nil
}
```

```
package main
import (
  "rpc/jsonrpc"
  "rpc"
  "os"
  "net"
  "log"
)

type RPCFunc uint8

func (*RPCFunc)
Echo(arg *string, result *string) os.Error {
  log.Print("Arg passed: " + *arg)
  *result = ">" + *arg + "<"
  return nil
}
```

```
package main
import (
  "rpc/jsonrpc"
  "rpc"
  "os"
  "net"
  "log"
)

type RPCFunc uint8

func (*RPCFunc)
Echo(arg *string, result *string) os.Error {
  log.Print("Arg passed: " + *arg)
  *result = ">" + *arg + "<"
  return nil
}
```

```
package main
import (
  "rpc/jsonrpc"
  "rpc"
  "os"
  "net"
  "log"
)

type RPCFunc uint8

func (*RPCFunc)
Echo(arg *string, result *string) os.Error {
  log.Print("Arg passed: " + *arg)
  *result = ">" + *arg + "<"
  return nil
}
```

```
func main() {
  log.Print("Starting Server...")
  l, err := net.Listen("tcp", "localhost:1234")
  defer l.Close()
  if err != nil {
    log.Fatal(err)
  }
  rpc.Register(new (RPCFunc))
  for {
    conn, err := l.Accept()
    if err != nil {
      log.Printf("accept error: %s", conn)
      continue
    }
    go jsonrpc.ServeConn(conn)
  }
}
```

```
func main() {
  log.Print("Starting Server...")
  l, err := net.Listen("tcp", "localhost:1234")
  defer l.Close()
  if err != nil {
    log.Fatal(err)
  }
  rpc.Register(new (RPCFunc))
  for {
    conn, err := l.Accept()
    if err != nil {
      log.Printf("accept error: %s", conn)
      continue
    }
    go jsonrpc.ServeConn(conn)
  }
}
```

```
func main() {
  log.Print("Starting Server...")
  l, err := net.Listen("tcp", "localhost:1234")
  defer l.Close()
  if err != nil {
    log.Fatal(err)
  }
  rpc.Register(new (RPCFunc))
  for {
    conn, err := l.Accept()
    if err != nil {
      log.Printf("accept error: %s", conn)
      continue
    }
    go jsonrpc.ServeConn(conn)
  }
}
```

```
func main() {
  log.Print("Starting Server...")
  l, err := net.Listen("tcp", "localhost:1234")
  defer l.Close()
  if err != nil {
    log.Fatal(err)
  }
  rpc.Register(new (RPCFunc))
  for {
    conn, err := l.Accept()
    if err != nil {
      log.Printf("accept error: %s", conn)
      continue
    }
    go jsonrpc.ServeConn(conn)
  }
}
```

```
func main() {
  log.Print("Starting Server...")
  l, err := net.Listen("tcp", "localhost:1234")
  defer l.Close()
  if err != nil {
    log.Fatal(err)
  }
  rpc.Register(new (RPCFunc))
  for {
    conn, err := l.Accept()
    if err != nil {
      log.Printf("accept error: %s", conn)
      continue
    }
    go jsonrpc.ServeConn(conn)
  }
}
```

```
func main() {
  log.Print("Starting Server...")
  l, err := net.Listen("tcp", "localhost:1234")
  defer l.Close()
  if err != nil {
    log.Fatal(err)
  }
  rpc.Register(new (RPCFunc))
  for {
    conn, err := l.Accept()
    if err != nil {
      log.Printf("accept error: %s", conn)
      continue
    }
    go jsonrpc.ServeConn(conn)
  }
}
```

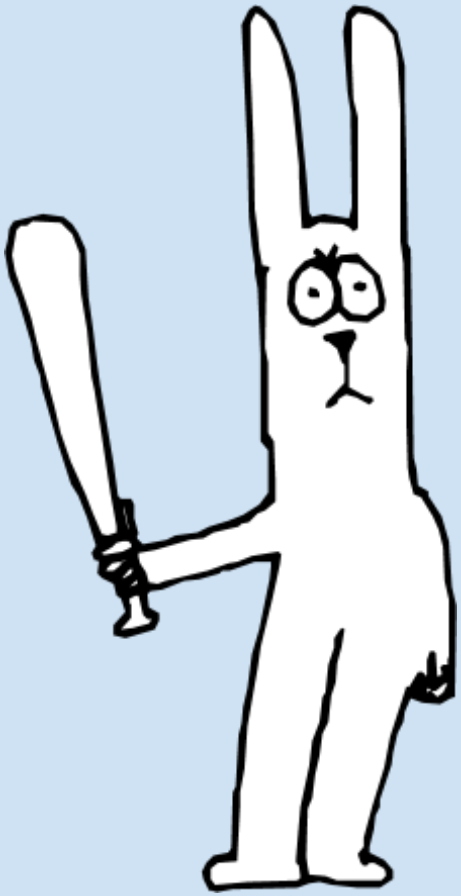
```
func main() {
  log.Print("Starting Server...")
  l, err := net.Listen("tcp", "localhost:1234")
  defer l.Close()
  if err != nil {
    log.Fatal(err)
  }
  rpc.Register(new (RPCFunc))
  for {
    conn, err := l.Accept()
    if err != nil {
      log.Printf("accept error: %s", conn)
      continue
    }
    go jsonrpc.ServeConn(conn)
  }
}
```

```
# JsonEchoClient.py
import jsonrpc # See articles for this

rpc = jsonrpc.ServerProxy(jsonrpc.JsonRpc10(),
        jsonrpc.TransportTcpIp(
            addr=("127.0.0.1", 1234)))

for i in range(10):
    print(rpc.RPCFunc.Echo("hello " + str(i)))
```

RPC Not Efficient Enough?



- Try it first, just to see (and to quick-test your app)
- If it's really a problem, consider Apache Thrift, which seems to be a "universal," efficient, language-to-language RPC
- There's also RPC with Google's Protocol Buffers

Hybridization example

If you don't believe hybridization is a good approach, consider this:

It's already happened with User Interfaces

The UI Problem

- Eventually users must see something; it would be nice to solve it once and for all.
- Flash is the most sophisticated, cross platform rapid-development UI solution I've seen
- Not dead yet
- What I would probably choose *until now*...

"HTML5"

- The defacto standard UI across devices -- especially handhelds
- At least it's a standard (but CSS is not great)
- Javascript is a terrible language, but not bad once the cross-platform idiocies are eliminated (jQuery & CoffeeScript)
- jQuery widgets probably do everything Flash widgets can, now.

The JavaScript Problem

- JavaScript is truly a mess.
- It has features you shouldn't use
- It hasn't had consistent syntax across browsers
- Browsers themselves have different DOM features and behave differently!
- You want to run screaming from it all...

What We Want

- "Write once, run everywhere"
- Never think about cross-browser differences
- Never think about whether you're using the language safely

Solution: jQuery + CoffeeScript

- jQuery: Browser-agnostic DOM manipulation
- Tons of jQuery plugins including jQuery UI
- Components that are matching or exceeding Flash

- CoffeeScript: Python-ish, Ruby-ish language, modern spare syntax with powerful constructs like comprehensions
- Write CoffeeScript, automatically generate correct, safe JavaScript
- BUT: Google's new Dart language for Web UIs

- The following example is described fully in this article (includes links to books and other resources).
- (I've added line breaks in the following slides which may not translate; use the article as a source reference).

```
<!doctype html>
<html>
 <head>
  <script type="text/javascript" src="jquery.js">
  </script>
  <script type="text/javascript" src="grid.js">
  </script>
 </head>
 <body>
  <p id="board" align="center"></p>
 </body>
</html>
```

```
<!doctype html>
<html>
 <head>
  <script type="text/javascript" src="jquery.js">
  </script>
  <script type="text/javascript" src="grid.js">
  </script>
 </head>
 <body>
  <p id="board" align="center"></p>
 </body>
</html>
```

```
<!doctype html>
<html>
 <head>
  <script type="text/javascript" src="jquery.js">
  </script>
  <script type="text/javascript" src="grid.js">
  </script>
 </head>
 <body>
  <p id="board" align="center"></p>
 </body>
</html>
```

```
cell = (row, col) ->
  "<td id='#{row}X#{col}'
   style='border-style:solid;border-width:8px;'>
   </td>"

row = (ncols, row) ->
  '<tr>' +
  (cell(row, col) for col in [0..ncols]).join('') +
  "</tr>\n"

table = (nrows, ncols) ->
  '<table>\n' +
  (row(ncols, n) for n in [0..nrows]).join('') +
  '</table>'
```

```
cell = (row, col) ->
  "<td id='#{row}X#{col}'
   style='border-style:solid;border-width:8px;'>
   </td>"

row = (ncols, row) ->
  '<tr>' +
  (cell(row, col) for col in [0..ncols]).join('') +
  "</tr>\n"

table = (nrows, ncols) ->
  '<table>\n' +
  (row(ncols, n) for n in [0..nrows]).join('') +
  '</table>'
```

```
cell = (row, col) ->
  "<td id='#{row}X#{col}'
   style='border-style:solid;border-width:8px;'>
   </td>"

row = (ncols, row) ->
  '<tr>' +
  (cell(row, col) for col in [0..ncols]).join('') +
  "</tr>\n"

table = (nrows, ncols) ->
  '<table>\n' +
  (row(ncols, n) for n in [0..nrows]).join('') +
  '</table>'
```

```
cell = (row, col) ->
  "<td id='#{row}X#{col}'
   style='border-style:solid;border-width:8px;'>
   </td>"

row = (ncols, row) ->
  '<tr>' +
  (cell(row, col) for col in [0..ncols]).join('') +
  "</tr>\n"

table = (nrows, ncols) ->
  '<table>\n' +
  (row(ncols, n) for n in [0..nrows]).join('') +
  '</table>'
```

```
randomRGBColor = ->
  val = -> Math.floor(Math.random()*255)
  "rgb(" + val() + "," + val() + "," + val() + ")"

randomize = (nrows, ncols) ->
  for r in [0..nrows]
    for c in [0..ncols]
      $('#' + "#{r}X#{c}")
      .css("border-color", randomRGBColor())
  setTimeout (-> randomize nrows, ncols), 1000

$ ->
  rows = 30; cols = 60
  $("#board").html(table rows, cols)
  randomize rows, cols
```

```
randomRGBColor = ->
  val = -> Math.floor(Math.random()*255)
  "rgb(" + val() + "," + val() + "," + val() + ")"

randomize = (nrows, ncols) ->
  for r in [0..nrows]
    for c in [0..ncols]
      $('#' + "#{r}X#{c}")
      .css("border-color", randomRGBColor())
  setTimeout (-> randomize nrows, ncols), 1000

$ ->
  rows = 30; cols = 60
  $("#board").html(table rows, cols)
  randomize rows, cols
```

```
randomRGBColor = ->
  val = -> Math.floor(Math.random()*255)
  "rgb(" + val() + "," + val() + "," + val() + ")"

randomize = (nrows, ncols) ->
  for r in [0..nrows]
    for c in [0..ncols]
      $('#' + "#{r}X#{c}")
      .css("border-color", randomRGBColor())
  setTimeout (-> randomize nrows, ncols), 1000

$ ->
  rows = 30; cols = 60
  $("#board").html(table rows, cols)
  randomize rows, cols
```

```
randomRGBColor = ->
  val = -> Math.floor(Math.random()*255)
  "rgb(" + val() + "," + val() + "," + val() + ")"

randomize = (nrows, ncols) ->
  for r in [0..nrows]
    for c in [0..ncols]
      $('#' + "#{r}X#{c}")
      .css("border-color", randomRGBColor())
  setTimeout (-> randomize nrows, ncols), 1000

$ ->
  rows = 30; cols = 60
  $("#board").html(table rows, cols)
  randomize rows, cols
```

```
randomRGBColor = ->
  val = -> Math.floor(Math.random()*255)
  "rgb(" + val() + "," + val() + "," + val() + ")"

randomize = (nrows, ncols) ->
  for r in [0..nrows]
    for c in [0..ncols]
      $('#' + "#{r}X#{c}")
      .css("border-color", randomRGBColor())
  setTimeout (-> randomize nrows, ncols), 1000

$ ->
  rows = 30; cols = 60
  $("#board").html(table rows, cols)
  randomize rows, cols
```

```
randomRGBColor = ->
  val = -> Math.floor(Math.random()*255)
  "rgb(" + val() + "," + val() + "," + val() + ")"

randomize = (nrows, ncols) ->
  for r in [0..nrows]
    for c in [0..ncols]
      $('#' + "#{r}X#{c}")
      .css("border-color", randomRGBColor())
  setTimeout (-> randomize nrows, ncols), 1000

$ ->
  rows = 30; cols = 60
  $("#board").html(table rows, cols)
  randomize rows, cols
```

```
randomRGBColor = ->
  val = -> Math.floor(Math.random()*255)
  "rgb(" + val() + "," + val() + "," + val() + ")"

randomize = (nrows, ncols) ->
  for r in [0..nrows]
    for c in [0..ncols]
      $('#' + "#{r}X#{c}")
      .css("border-color", randomRGBColor())
  setTimeout (-> randomize nrows, ncols), 1000

$ ->
  rows = 30; cols = 60
  $("#board").html(table rows, cols)
  randomize rows, cols
```

```
randomRGBColor = ->
  val = -> Math.floor(Math.random()*255)
  "rgb(" + val() + "," + val() + "," + val() + ")"

randomize = (nrows, ncols) ->
  for r in [0..nrows]
    for c in [0..ncols]
      $('#' + "#{r}X#{c}")
      .css("border-color", randomRGBColor())
  setTimeout (-> randomize nrows, ncols), 1000

$ ->
  rows = 30; cols = 60
  $("#board").html(table rows, cols)
  randomize rows, cols
```

```
randomRGBColor = ->
  val = -> Math.floor(Math.random()*255)
  "rgb(" + val() + "," + val() + "," + val() + ")"

randomize = (nrows, ncols) ->
  for r in [0..nrows]
    for c in [0..ncols]
      $('#' + "#{r}X#{c}")
      .css("border-color", randomRGBColor())
  setTimeout (-> randomize nrows, ncols), 1000

$ ->
  rows = 30; cols = 60
  $("#board").html(table rows, cols)
  randomize rows, cols
```

```
randomRGBColor = ->
  val = -> Math.floor(Math.random()*255)
  "rgb(" + val() + "," + val() + "," + val() + ")"

randomize = (nrows, ncols) ->
  for r in [0..nrows]
    for c in [0..ncols]
      $('#' + "#{r}X#{c}")
      .css("border-color", randomRGBColor())
  setTimeout (-> randomize nrows, ncols), 1000

$ ->
  rows = 30; cols = 60
  $("#board").html(table rows, cols)
  randomize rows, cols
```

```
randomRGBColor = ->
  val = -> Math.floor(Math.random()*255)
  "rgb(" + val() + "," + val() + "," + val() + ")"

randomize = (nrows, ncols) ->
  for r in [0..nrows]
    for c in [0..ncols]
      $('#' + "#{r}X#{c}")
      .css("border-color", randomRGBColor())
  setTimeout (-> randomize nrows, ncols), 1000

$ ->
  rows = 30; cols = 60
  $("#board").html(table rows, cols)
  randomize rows, cols
```

```
randomRGBColor = ->
  val = -> Math.floor(Math.random()*255)
  "rgb(" + val() + "," + val() + "," + val() + ")"

randomize = (nrows, ncols) ->
  for r in [0..nrows]
    for c in [0..ncols]
      $('#' + "#{r}X#{c}")
      .css("border-color", randomRGBColor())
  setTimeout (-> randomize nrows, ncols), 1000

$ ->
  rows = 30; cols = 60
  $("#board").html(table rows, cols)
  randomize rows, cols
```

HTML5 Can Create UIs for Apps

- Python Server
  - Should probably use Django for anything big and fancy
  - But let's go simplest: Web.py (quick to understand)

- CoffeeScript+jQuery UI
  - Use Ajax so it works like an app instead of jumping to pages.
  - Communicates both ways with server, refreshes itself
- Here's the detailed article about this example. Article also shows how app can be deployed on Heroku.

This works, but...

- Polling the server can be too primitive
- HTML5 is standardizing Server-Sent Events
- Supported now in most browsers (not IE9, maybe 10)
- Qualifies HTML5 as a full UI
- Tornado (feels like Web.py, handles tons of connections) has a Websocket library; the resulting code seems much simpler.

HTML5 Components are Amazing

Example: JSXGraph

Upcoming:

- Java Posse Roundup, March 26-30, 2012, Crested Butte Colorado. (See great photos of past Roundups)
- Scala Summer Camp, sometime during the summer of 2012, Crested Butte, CO.

Links: RPC

- Go Language (comes with JSON-RPC library)
- JSON-RPC (1.0/2.0) Library for Python
- JSON-RPC Plugin for JQuery
- Apache Thrift
- Thrift for Go (also here)
- JSON-RPC JavaScript Client Library

Links: JavaScript

- Eloquent JavaScript: Very well-written free introductory book
- CoffeeScript
- CoffeeScript + JQuery is part of Rails 3.1
- My article on the problems of JavaScript

Links: UIs in HTML5/JavaScript/CSS

- Python web.py simplest web framework
- JQuery tutorials
- Article introducing web.py + JQuery
- Skeleton Support for Browsers & Handhelds
- Django
- JSON-RPC JavaScript Client Library
- Balsamiq UI Prototyping tool
- Amazing JSXGraph Javascript Graphing Library (also supports iOS and Android devices)

- Scala Koans
- Twitter's Bootstrap Styles-to-get-started
- My articles on hybridizing Python with Flex UIs

Questions?

Or a discussion session in the lobby later...