Scala and Go: A Comparison of Concurrency Features

Brent Smith and Leon Gibson
CSCI5828 Spring 2012

Comparison of Scala & Go's concurrency constructs

By Brent Smith and Leon Gibson

What does Scala and Go offer your company for a scalable concurrent solution?

- Scala
  - Immutable, persistent data structures
  - Functional Programming via First class functions, and Closures
  - Scales up (concurrency) and out (remoting) with the Actor model
  - Software Transactional Memory
- Go
  - An expressive lightweight machine code driven language
  - Re-introduces the concept of "goroutines" and "Unix pipe-like" channels
  - Simple abstractions that support concurrency via isolated mutability
  - High speed compilation

Agenda

- Introduction to languages
- Discuss Scala concurrency features
- Discuss Go concurrency features
- Examples included in ScalaAndGo-CSCI5828_S12_BSMITH-LGIBSON.zip
- Comparison and Summary

Scala

- General purpose language
- Built on top of the Java VM platform
- Mixes OO (imperative) and functional programming styles
- Syntax is more concise and extensible than Java
- Average 2x reduction in code size vs. Java programs
- Scales up (concurrency) and out (remoting)
- Built-in support for Actors based concurrency model

Scala and Java

- Fully compatible with Java
  - Java code can be used from Scala, and vice versa
  - Compiles to .class files
  - Scala is essentially another .jar library

- Designed by Martin Odersky
  - Contributed to the current generation of the **javac** compiler
  - Founder of Typesafe, which provides enterprise level support for a software stack that consists of Scala, Akka, and other libraries

Scala in a nutshell

- Is more object-oriented than Java, as *everything* is an object
  - primitive types removed
  - functions are first class objects
  - of course, classes are still around
  - static methods have been removed
- Java interfaces are replaced by **traits**
  - which more closely resemble abstract classes
- Has its own type hierarchy
  - All objects derived from the **scala.Any** class
- Supports closures
- Mixin class composition (akin to *multiple inheritance*)

Installing Scala

- Download distribution from http://www.scala-lang.org/
- Extract archive
- Ensure **JAVA_HOME** environment variable is set and **$JAVA_HOME/bin** is in your **PATH**
- Set **SCALA_HOME** to directory where you extracted Scala
- Add **$SCALA_HOME/bin** to your **PATH**
- Run **scala** to get an interactive interpreter
- Run **scalac** to compile a .scala file
- We will download a 3rd party library called Akka later when we discuss Actors

Obligatory Example

```
object Hello extends App {
       var msg = "Hello, World! "
       println(msg + (args mkString ":") )
}
```

Compile and run as follows:

```
$ scalac -cp $SCALA_HOME/lib/scala-library.jar Hello.scala
$ java -cp $SCALA_HOME/lib/scala-library.jar:. Hello Brent
and Leon!
```

Prints:

```
Hello, World! Brent:and:Leon!
```

From Java to Scala

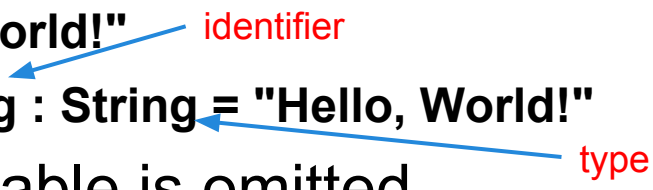
- For Java programmers, it may not be exactly straightforward to pick up Scala
  - There are certain features that make the syntax of the language differ significantly from Java.
  - These features are:
    - Various forms of "Syntactic Sugar"
    - Local Type Inference
    - First Class Functions and Closures
    - Operator overloading
- Since the main topic of the presentation is concurrency features, we only discuss the above issues briefly on the next few slides

"Syntactic Sugar"

- In our Hello, World! example, we wrote the following
  - **println(msg + (args mkString ":"))**
- Dot operator and parenthesis are optional for methods
  - Equivalent code is **println(msg + args.mkString(":"))**
  - Since we omitted the dot operator, we need to add parentheses around (args mkString ":"), otherwise **msg + args** is evaluated first
- Other forms of syntactic sugar
  - Return statement is optional. If omitted, return value is inferred from the last line of a method
  - Ternary operator: **if (cond) expr1 else expr2**
  - The **object** keyword (instead of **class**) implements singleton pattern
  - The constructor parameters can be given in the definition
  - code blocks can omit the curly braces { } if the block consists of only a single statement/expression

Local Type Inference

- In our Hello, World! example, we wrote the following
  - **var msg = "Hello, World!"**
  - equivalent to **var msg : String = "Hello, World!"**



- The type of the variable is omitted
  - Type is *String*
  - Compiler can determine the type of the **msg** variable from the literal on the right-hand-side
  - Return type from functions/methods can typically be omitted since the type can be inferred from the body of the function
- In some cases, the type must be specified
  - Recursive methods must specify a type
- Scala is still a statically typed language!

First Class Functions

- Functions are objects too, and therefore can be assigned to variables and passed as arguments as functions
- But first, it's important to understand the anonymous function syntax:
  - This is an anonymous function taking an Int and returning an Int
    - **(x : Int) => x + 1**
  - The return type is implicit above, but can also be specified as follows
    - **(x : Int) => x + 1 : Int**
  - Anonymous function taking two Ints and returning a String
    - **(x : Int, y : Int) => "(" + x + "," + y + ")"**
  - Anonymous function taking no arguments and returning a String
    - **() => "Hello!"**

First Class Functions (Example)

- Assign an anonymous function to variable **square**

```
scala> var square = (x : Int) => x*x
square: Int => Int = <function1>
```

- Use the **map** function on a **List** to generate a new collection by applying a function to all elements of a list

```
scala> List(1,2,3,4).map(square)
res2: List[Int] = List(1, 4, 9, 16)
```

- Define a function **f** that takes an **Int** and another function **g**

```
scala> def f(x : Int, g : (Int)=>Int) = g(x+1)
f: (x: Int, g: Int => Int)Int
scala> f(1, square)
res0: Int = 4
```

Closures

- In addition to anonymous functions, Scala allows closures
  - allow you to define an anonymous code block that *closes* over a free variable
  - A free variable is a variable defined in the parent or ancestor block
  - Scala does *dynamic binding* of the free variable
    - If the variable changes a value at a later time, then the closure will use the updated value
- Example

```
var msg = "Hello"
def f = { println(msg) }
f ; msg = ", World!" ; f
```

- Prints:

```
Hello
, World!
```

Operator Overloading

- Scala allows method names to be operators
  - When combined with the "syntactic sugar" we saw for method calls, where the dot operator and parentheses can be omitted, this makes a method call look like standard infix notation
  - Important later when we see how to send messages to Actors
  - Example:

```
scala> class IntWrapper(value : Int) {
     |   def + (other : Int) = new IntWrapper(value + other)
     |   override def toString = value.toString
     | }
scala> var a = new IntWrapper(1)
a: IntWrapper = 1
scala> a = a + 1
a: IntWrapper = 2
scala> a = a.+(2)
a: IntWrapper = 4
```

Define a class called IntWrapper, that wraps an Int and overrides the method "+" and "toString"

Create an instance of IntWrapper

Method call to + method using what looks like standard infix notation

Method call to + method using standard method call notation

Concurrency in Scala

- Scala has several desirable features for concurrent applications
  - Many data types are immutable by default
  - Provides useful abstractions for easily doing thread based concurrency
  - Provides support for functional programming and first class functions
    - First class functions simplify certains tasks (instead of creating anonymous inner classes that implement interfaces, just create an anonymous function)
    - Pure functional programming has no side-effects therefore lends itself to concurrency
      - Hadoop is an example where functional programming (map/reduce) is used to perform concurrent computation
  - As of Scala 2.9, has support for parallel collections
  - Software Transaction Memory is available through additional libraries
    - Akka Actors, Akka STM, ScalaSTM
  - Actor-based model is provided out-of-the-box

Immutable Types

- All immutable types in scala are annotated with the marker trait **scala.Immutable**
  - Includes Tuple2, List, Map
  - Some types are implemented with efficient "persistent" data structures (tries), which makes copying an immutable collection class cheap/fast
  - Operators such as +: are overloaded for some collections, and invoking these creates a new collection efficiently
- Examples of creating immutable types
  - `scala> List("Hello",1,'a')`
  - `res0: List[Any] = List(Hello, 1, a)`
  - `scala> var b = Tuple2(List(1,2),3)`
  - `b: (List[Int], Int) = (List(1, 2),3)`
- Immutable types are required when passing messages to Actors (we'll see Actors soon)

Useful Abstractions: ThreadRunner

- ThreadRunner wraps java.util.concurrent. ExecutorService
  - Can simply submit an anonymous function instead of implementing **Runnable** or **Callable**
  - Automatically returns a function that, when called, returns the value that would typically be returned by Future.get()
- Example

```
scala> var threadRunner = new ThreadRunner()
scala> threadRunner.execute(() => { println("Hello from " + Thread.
currentThread().getName()) } )
Hello from Thread-34
scala> var future = threadRunner.submit(() => { "Hello from " + Thread.
currentThread().getName() } )
future: () => java.lang.String = <function0>
scala> var a = future()
a: java.lang.String = Hello from Thread-36
```

Useful Abstractions: Future

- Scala provides a enhanced Future abstraction
  - Makes it even easier to submit a task to a thread pool
- Example

```
scala> import scala.actors.Futures._
scala> val f1 = future { Thread.sleep(5000); "Hello from "+Thread.
currentThread.getName()}
f1: scala.actors.Future[java.lang.String] = <function0>
scala> f1()
res5: java.lang.String = Hello from ForkJoinPool-1-worker-2
```

Import all the methods from the Futures singleton into the current namespace

Create a Future [String] that sleeps for 5 seconds and returns the thread name. Assign it to the f1 variable.

Wait for the result from the Future f1

- Features
  - Returns a first class function, that returns a value
  - When the first class function is invoked via (), it will block the current thread until the result is available
  - Uses a ForkJoin thread pool by default

Parallel Collections

- As of Scala 2.9, calling the **par** method on a collection that has the trait **CustomParallelizable** will return a parallel version of that collection
  - Methods on the collection will be automatically parallelized where possible (e.g., span is not parallelizable, but **filter**/**map**/**reduce** are)
  - Must be careful not to access mutable state (don't pass in a closure over a mutable free variable)
- Uses ForkJoin thread pool under the hood
  - As a consequence, no control over size of the pool
    - Pool is typically equal to # processors, but can grow to prevent deadlock/starvation
  - Get load balancing of tasks for free due to work stealing
- Best for CPU intensive tasks
- May not produce good results for I/O intensive tasks

Parallel Collections (Example)

```
def time(f : () => Unit) = {
  val t1 = System.currentTimeMillis()
  f()
  println("Elapsed: " + ((System.currentTimeMillis() - t1)/1.0e3));
}
def isPrime(x : Int) : Boolean = {
  for (i <- 2 until x) {
    if ((x % i) == 0) {
      return true
    }
  }
  return false
}
scala> time(() => { (1 to 100000).par.filter(isPrime) } )
Elapsed: 1.373
scala> time(() => { (1 to 100000).filter(isPrime) } )
Elapsed: 3.871
```

Function that takes another function as an argument, runs the function and prints out the elapsed time

Naive function to compute whether a given number is a prime (Yes, there are more efficient ways to compute a prime)

Use **par** to return a parallel version of the collection, and call the filter(isPrime) to compute all primes between 1 and 100,000. The execution time is significantly faster than the "sequential" version of the collection (this test was done on a 4 core machine)

Software Transaction Memory (STM)

- Popularized by Clojure
  - Lisp like language that also runs on the JVM
- Attempts to separate *identity* and *state*
  - state is an immutable *value* that a reference variable points to
  - identity is mutable and simply switches between immutable states
  - changes in state create a new *value* (old value persists in memory)
  - eliminates costly locking overhead
- Supports atomicity, consistency and isolation (ACI)
  - transactions ensure that changes in state either commit or rollback
  - consistency ensures that the state never violates invariants after a commit or rollback
  - isolation ensures that changes done in a transaction are only visible from within that transaction (and not transactions on other threads)
  - no durability, since this is all done in memory

STM in Scala

- STM is provided via Akka library
  - Uses ScalaSTM library, which will soon be included in Scala
- To use it
  - You'll need the following in your classpath:
    - $AKKA_HOME/lib/akka/scala-stm_2.9.1-0.5.jar
  - **import scala.concurrent.stm._**
  - Wrap your mutable variables that you wish to share in a **Ref**
  - Define a transaction using the **atomic** keyword
  - read a **Ref** by appending () after the identifier representing it, e.g., x()
  - write a **Ref** by appending () and then assigning it a value; e.g. x() = 1
- Basic example provided as we'll cover more in Prof. Anderson's lecture

STM Example

```
import scala.concurrent.stm._

object Main extends App {
  val x = Ref(0)
  atomic { implicit txn =>
      if((x() % 2) == 0){
          println("x is even: " + x())
          retry
      }
  } orAtomic { implicit txn =>
    x() = x() + 1
    println("x is currently: " + x())
  }
}
```

Ref-s can only be accessed within an atomic block.

Creating an atomic function requires the passing in of the InTxn type; implicit txn satisfies this formality

orAtomic functions execute when an atomic function ends in a retry

Obtain the reference to x and set its value equal to x() + 1

Output:

x is even: 0

x is currently: 1

Actors-based Concurrency Model

- Popularized by Erlang, and now by Scala
- Actors are "light-weight" processes that
  - Encapsulate state
  - Send/Receive messages
  - Have a *mailbox* to store unprocessed messages
  - Guaranteed to be scheduled on at most one thread for execution, and thus do not need synchronization
  - Many more actors can exist than threads
  - Potentially scale to millions of actors
- Simplifies concurrent programming via isolated mutability
  - State is never shared between actors

Actors in Scala

- Scala has a built-in library for Actors
- There are also 3rd party libraries, such as Akka, that provides Actors as well
  - Eventually the Akka actors will be rolled into Scala
  - So, we will focus on the Akka 2.0 implementation of actors
  - Part of the package **akka.actor**
- Rules for Actors
  - All messages passed to Actors *must be immutable!*
  - Actors should not attempt to access any mutable state that is not encapsulated by an Actor
    - Actors can only send messages to other Actors
    - Messages must not contain closures that close over a non final free variable
    - Right now, scala cannot enforce the constraint of immutability, so the programmer has to ensure that this is the case.

Sending Messages to Actors

- In Akka 2.0, two ways you can send messages to Actors
- *Tell* or *Fire-and-forget*
  - Achieved via the overloaded ! operator
  - Example:

    actorToSendMsg ! msg
- *Ask* (get back a Future)
  - Achieved via the overloaded ? operator
  - Example:

    var future = actorToSendMsg ? msg
  - Actor receiving the **msg** has to send a reply back using the special **sender** variable, and the usual **!** method
  - Example:

    sender ! replyMsg

Creating an Actor

- To create an actor in Akka 2.0
  - Define a class and extends the **akka.actor.Actor** trait
  - Define a **receive** function (takes no arguments)
  - Use a **case** statement to perform different actions based on the message received
    - Case statement can take a *type* or a *value*, but must be immutable.
    - Special *case classes* are used for parameterized messages.
- Ping Pong example next
  - Create a **Server** actor which receives a *Ping* message and replies with a *Pong*
  - Create a **Client** actor which receives a *Start* message and responds by sending a *Ping* message to the server
  - Main program which creates the Actors and sends a *Start* to the client

PingPong! Server

```
class Server extends Actor {
  def receive = {
    case "Ping" => {
      println(Thread.currentThread().getName() + ": 
Ping!")
      sender ! "Pong"
    }
    case _ => println("Unknown message!")
  }
}
```

Extend **Actor** trait

Define **receive** method

Use **case** statement to handle "Ping" message

Infix notation for the overloaded ! operator which means: *send the String "Pong" to whoever sent us the "Ping" message*.

Handle any messages that don't match prior rules. Similar to the **default** keyword in the typical Java switch statement.

PingPong! Client

```
import akka.actor._
class Client(server : ActorRef)  extends Actor {
  def receive = {
    case "Start" => server ! "Ping"
    case "Pong" => {
      println(Thread.currentThread().getName() + ":
Pong!")
      context.stop(server)
      context.stop(self)
      context.system.shutdown
    }
    case _ => println("Unknown message!")
  }
}
```

Import everything directly under the package akka.actor

Stop the server and client Actors.

Shutdown the **ActorSystem**. The system that this actor belongs to, is available from the **context** variable within an Actor. This is necessary to shutdown the underlying thread pool. If we don't do this, then the threads in the thread pool will prevent the JVM from shutting down, since these are non-daemon threads by default.

PingPong! Main

```
object Main extends App {
  val system = ActorSystem("PingPong")
  val server = system.actorOf(Props[Server])
  val client = system.actorOf(Props(new Client(server)))
  client ! "Start"  // get the ball rolling
}
```

- Output

```
PingPong-akka.actor.default-dispatcher-2: Ping!
PingPong-akka.actor.default-dispatcher-2: Pong!
```

Notice that the Client and the Server are both using the same thread #2! Each actor got scheduled on the same thread. In other words, the scheduling system can multiplex actors onto a single thread, as long as certain conditions are true (we'll see what these are in a second). Run this multiple times and you should see each Actor running on the same thread consistently.

PingPong! with a twist

- With just a small, seemingly insignificant change in the code, we can affect the output

```
class Server extends Actor {
  def receive = {
    case "Ping" => {
      sender ! "Pong"
      println(Thread.currentThread().getName() + ":
Ping!")
    }
    case _ => println("Unknown message!")
  }
}
```

- Can you spot the change?

PingPong! with a twist (2)

- We just switched the order of the **println** and the **sender ! "Pong"** statements from the original

- How does this affect the output?
  - The "Ping" and "Pong" messages get printed from different threads now!

    ```
    PingPong-akka.actor.default-dispatcher-1: Ping!
    PingPong-akka.actor.default-dispatcher-3: Pong!
    ```

  - This should be consistent; if not, try adding a Thread.sleep(100) immediately after the **println**

- Why does the output/behavior change?

PingPong! with a twist (3)

- If you guessed because **println** blocks, you'd be correct
- The thread assigned to the Server actor can't be returned to pool until code in **receive** function completes
- 1st case: **println** *before* sending the "Pong" message
  - The "Pong" message was queued in the mailbox for the Client actor, then the thread assigned to the Server actor was immediately returned to the thread pool.
  - The same thread was then available to service the Client actor and its pending "Pong" message
- 2nd case: **println** *after* we send the "Pong" message
  - The "Pong" message was queued in the mailbox for the Client actor, then the thread assigned to the Server actor had to execute the println method, which blocks (writes I/O to console)
  - Since the Server actor's thread is busy blocking on I/O, another thread is assigned to the Client actor to handle the **Pong** message

Best Practices

- Try to avoid blocking operations in your Actors
  - This is very hard to do.
  - Most database libraries require blocking I/O
  - Many of Java's standard libraries rely on blocking I/O
    - However, the java.nio package supports asynchronous I/O
- If you can't avoid blocking I/O, then use 2 thread pools
  - One for non blocking actors (CPU bound)
  - One for blocking actors (I/O bound)
  - The reason for this is so you can configure the size of the pools in an intelligent way
    - With a single pool, you may unintentionally limit the number of Actors that can be serviced if all of your threads are blocking on I/O
  - Use Akka Dispatchers to associate thread pools with Actors
    - See http://doc.akka.io/docs/akka/2.0/scala/dispatchers.html

Typed Actors

- Turn method invocations into asynchronous messages
  - Uses the Active Objects pattern and a proxy to do this
- How it's done in Akka
  - Define an interface (trait) and corresponding implementation
  - Extend **akka.actor.TypedActor** trait instead of **Actor**
  - Define methods on interface
  - Define corresponding implementation
- Method semantics defined by return value
  - Returns **Unit** (equivalent to void) => *fire and forget*
  - Returns **Future[T]** => Same as *ask* (non blocking request-reply)
  - Everything else => Blocking request reply
- Instantiate a TypedActor
  - use **TypedActor(*sys*).typedActorOf(...)** (*sys* is an ActorSystem)

Typed Actors (2)

- See the included examples for a Typed Actors version of the Ping Pong server
- Difficulties we ran into...
  - The sender is not available from within the Server, so we have to encode the client in the ping(client:Client) method, so that the server can call client.pong()
  - We got runtime exceptions when we didn't follow the rules exactly
    - The return type from TypedActor(system).typeActorOf must be explicitly specified as the interface, or a runtime ClassCastException is thrown
    - When calling TypedActor.context.stop(), need to pass TypedActor.context.self instead of TypedActor.self
      - The former is a reference to the ActorRef
      - The latter is a reference to a Proxy

Go

- Developed by Google Inc. in 2007 and officially announced in 2009
  - Written by Robert Griesemer, Rob Pike, and Ken Thompson.
  - Said to be used "for real stuff" at Google.
- Supports ease of use while being efficient as a statically-typed compiled language.
  - Both type-safe and memory-safe
  - Supports concurrency and communication through channels
  - Efficient garbage collection
  - High speed compilation

Basics

- Source is UTF-8
  - No semicolons or rules related to tabs or spacing like Python
  - Parenthesis are not required except for argument lists
- Includes all familiar types including support for int8, uint32, float64, etc.
- Strings are immutable
- Only control statements are `if`, `for`, `switch`, and `defer`
- All memory in Go is initialized (un-initialized variables are of "zero value")

Compiling

- Runs on the i386, amd64, and ARM architectures with the "gc" compilers
  - 8g <= i386
  - 6g <= amd64
  - 5g <= ARM
- Alternatively a gccgo compiler is available for use with traditional gcc
- Compiling your first file

```
$ 8g testFile.go
$ 8l testFile.8
$ ./8.out
```

Declarations

- Declarations are reversed with type at end
  - `var x int`
  - `var j = 365.245`
  - `var k int = 0`
- Multiple assignment fun
  - `var l, m uint64 = 1, 2`
  - `var inter, floater, stringer = 1, 2.0, "hi"`
  - `var ( x, y, z = 42, "Hello", f3() )`
- Shorthand declarations (within functions only)
  - `i := "Hello"`
- Function declarations
  - `func f1(){}`

Notable Nuances

- For is the only loop structure
  - `for {}`
  - `for a{}`
  - `for ;;;{}`
  - `for x := range a {}`
- Switch statements
  - Expressions do not need to be a constant or an int, and multiple cases can be comma-seperated.
  - No automatic fall through
- Functions can return multiple values
- More useful Go features: slices, defer, iota, and the blank identifier "_"

Technical Notes

- No support for Generics/Templates
  - Instead relies on built in maps, slices and explicit unboxing to provide similar functionality
- Exceptions are not included
  - `os.Error` variable represents any value that can describe itself as a string.
- Type inheritance is not supported
  - Types automatically satisfy any interface that specifies a subset of its methods.
- Garbage collection is performed by mark-and-sweep

Goroutines

- A function that executes in parallel with other goroutines in the same address space
  - Play on the word "coroutine"
  - Not to be confused with a thread, process, or actual coroutines.
- If a goroutine becomes blocked the runtime scheduler switches in another goroutine to the thread for execution.
- Prefix the word "go" to any function and it becomes a goroutine

Goroutines (cont.)

- Each goroutine shares the same memory space within a program
  - Programmer need not worry about the stack
  - Tests have shown tha each goroutine uses 4-5kB per stack address.
  - Uses the heap to allocate and free more space for the goroutine stack.
- Communication between goroutines exists in the use of channels

Goroutine Scheduling

- With the gc Go compiler (6g or 8g) all goroutines multiplex using one OS thread
  - Using the shell var GOMAXPROCS the number of cores can be specified resulting in > 1 threads
  - A thread will be created per GOMAXPROCS but Goroutine channel performance suffers from the resultant context switching
    - "In future, [goroutine scheduler] should recognize such cases and optimize its use of OS threads." -- golang.org

- The gccgo compiler uses one OS thread per goroutine
  - Performance can actually improve with GOMAXPROCS > 1

Channels

- Very similar to UNIX pipes as it enables synchronization between goroutines
- Allocating a channel is as easy as...
  - `ch1 := make(chan int)`
- Communication with the **<-** operator.
  - `v = <-c  // receive value from c, assign to v`
  - `<-c  // receive value, throw it away`
  - `i := <-c  // receive value, initialize`
  - `var recvChan <-chan int  //Receive only channel`
  - `var sendChan chan<- int  //Send only channel`

"Do not communicate by sharing memory; Instead, share memory by communicating"

Channels

- Synchronous communication (unbuffered)
  - A channel operation blocks until there is a matching operation on the other end.
    - A send operation requires a receive to complete on the channel otherwise it will block.
    - Likewise a receive operation blocks until there is a send operation on the same channel.
- Asynchronous communication (buffered)
  - Use the **make** keyword passing in an integer size value
    - `var bufChan = make(chan, 10)`

Simple Channel Example

```
package main
import fmt "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

Closures

- Just as in Scala, golang allows for closures over a variable in an inner function
- Let's create an example function called who()
  - Notice that who() takes an inner unnamed func(string) that returns a string
  - A var noun is defined as a string outside of the inner func(name string)

```
func who() (func(string) string) {
    var noun string
    return func(name string) string {
        noun += name
        return noun
    }
}
```

  - Adding this closure to our channel example provides a useful example...

Channels using who() closure

- `sayHello() now creates a variable f that equals function who()`[1]

```
func sayHello (ch1 chan string){
  var f = who()
  ch1<-"Hello " + f("Brent & Leon\n")
}
```

$ 8g closureHello.go ; 8l -o closureHello closureHello.8

$ ./closureHello

Hello Brent & Leon

[1] - main() does not need a change

No STM?

- With Go channel synchronization there is no need for STM
  - This leads to a highly scalable solution without the resources needed for expensive transactions
- "Happens Before" rules within a goroutine
  - Compilers and processors may reorder the reads and writes if it does not change the behavior of the goroutine.
  - Goroutine B may "perceive" a different order of actions in goroutine A.

"If event e_1 happens before event e_2, then we say that e_2 happens after e_1. Also, if e_1 does not happen before e_2 and does not happen after e_2, then we say that e_1 and e_2 happen concurrently"

Go: Ping Pong Example

- Let's recreate the ping pong example using Go
  - Create a **Server** function which receives a *Ping* message over a channel and replies with a *Pong*
  - Create a **Client** function which receives a *Start* message and responds by sending a *Ping* message to the server
  - A Main program which starts the client and server as goroutines and sends a *Start* to the client using a channel

Go: PingPong! Client

```
func client(hbChan chan string, quitChan chan<- string){
    hbChan <-"Ping"
    switch msg := <-hbChan;{
        case msg == "Pong":
            fmt.Printf("Client: Got my Pong!\n")
        default:
            fmt.Printf("Unknown message")
    }
    quitChan <- "Done\n"
}
```

Send a message to the Server

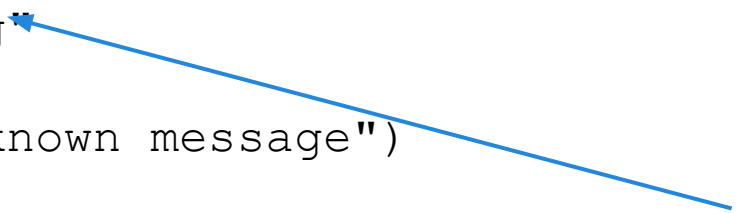
Declare a "Send-only" channel argument

Declaring a variable in our switch statement equal to the first unblocked read of our channel

Signal to our quit channel that the client has completed processing

Go: PingPong! Server

```
func server(hbChan chan string){
    switch msg := <-hbChan; {
        case msg == "Ping":
            fmt.Printf("Server: Received a Ping!\n")
            hbChan <- "Pong"
        default:
            fmt.Printf("Unknown message")
    }
}
```



Use the bi-directional channel to receive both the "Ping" and send the "Pong" to the channel reciever

Go: PingPong! Main

```
func main() {
    var hbChan = make(chan string)
    var quitChan = make(chan string)
    go server(hbChan)
    go client(hbChan, quitChan)
    fmt.Printf(<-quitChan)
}
```

Declare two channels

Start each function as goroutines and pass the channel communication as well as the channel to the client to complete the program

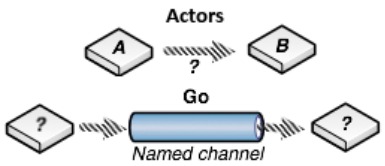
Output
$ ./pingpong
Server: Received a Ping!
Client: Got my Pong!
Done

No need for a timer or sleep function, just use the quit channel to allow a client the opportunity to declare when processing is complete

An improvement to this program would be an array of heartbeat channels passed to the server with each client receiving one hearbeat channel each. Therefore allowing the program to scale.

Scala vs. Go - Concurrency Comparison

- Both languages support first class functions and closures
- Scala offers some features that Go does not, at the cost of additional complexity and a higher learning curve
  - Immutable, persistent data structures
  - *ThreadRunner* and *Future* abstractions
  - Parallel collections
  - Software transactional memory via Akka / ScalaSTM
- Both have frameworks that support "lightweight" processes that are multiplexed on underlying threads
- But they way they communicate is different
  - **Scala**: *actors* send messages to other *actors*
  - **Go**: *goroutines* send/receive messages on *channels*



Scala vs. Go - Concurrency Comparison (2)

- As always, evaluate your options and use the best tool for the job
  - Scala runs on a JVM. This means its heavier, but can integrate with legacy code, and use existing Java libraries. This may be enough to drive your decision.
  - Go compiles to platform dependent code on a relatively few number of platforms, but is simpler and a clean solution to many concurrent problems. However, the language is fairly young, still only supports a primitive mark-and-sweep garbage collector and may lack library support.
  - Both have "light-weight" processes: Actors in Scala, and goroutines in Go.
  - Only Scala has 3rd party support for STM.
    - STM should only be used for highly concurrent reads, but infrequent writes. This is applicable for some applications (think Amazon shopping cart) but not others

Conclusions

- Scala is built on top of the JVM
  - Get Java libraries for free and 100% compatible with Java
- Scala supports concurrency through a variety of built-in classes and 3rd party frameworks
  - Immutable, persistent data structures
  - Functional programming / First Class Functions
  - Useful abstractions (ThreadRunner, Future and others...)
  - Parallel Collections
  - Software Transactional Memory
    - form of shared mutability, but separates *identity* and *state*
    - Doesn't scale well with concurrent writes
  - Support for Actor-based concurrency
    - form of isolated mutability
    - Untyped/typed actors and different message type (*tell*, blocking/ non-blocking *ask*)

Conclusions (2)

- Go is "closer to the metal"
  - Compiles to platform dependent code (amd64, i386)
  - Syntax is simple (feels like a mix of C/Java)
  - Partially object oriented, but no support for some features such as inheritance
  - Supports safe pointers
- Go provides concurrency in the form of **goroutines** and **channels**
  - **goroutines** are lightweight background tasks that are multiplexed on underlying threads from a thread pool
  - **channels** allow communication between **goroutines** in a thread safe manner, similar to a BlockingQueue or UNIX pipes
  - No support for software transactional memory

References

- `golang.org`
  - http://golang.org/doc/go_for_cpp_programmers.html
  - http://golang.org/doc/effective_go.html
  - http://golang.org/doc/GoCourseDay1.pdf
  - http://golang.org/doc/GoCourseDay2.pdf
  - http://golang.org/doc/GoCourseDay3.pdf
  - http://golang.org/doc/go_mem.html
- Golang: goroutines performance
  - http://en.munknex.net/2011/12/golang-goroutines-performance.html
- GoLang Tutorials: Goroutines
  - http://golangtutorials.blogspot.com/2011/06/goroutines.html

References (2)

- How To Achieve Concurrency In Google Go
  - http://code.rkevin.com/2010/10/how-to-achieve-concurrency-in-google-go/
- `Scala-lang.org`
  - A Tour of Scala - http://www.scala-lang.org/node/104
  - Scala Actors - http://www.scala-lang.org/node/242
  - The Scala Actors API - http://docs.scala-lang.org/overviews/core/actors.html
- `Akka.io`
  - Akka Documentation - http://doc.akka.io/docs/akka/2.0/
  - Akka and the Java Memory Model - http://doc.akka.io/docs/akka/1.3.1/general/jmm.html
- Concurrency with Actors, Goroutines and Ruby
  - http://www.igvita.com/2010/12/02/concurrency-with-actors-goroutines-ruby/

References (3)

- Scala STM - Quick Start
  - http://nbronson.github.com/scala-stm/quick_start.html
- GPars – Actors
  - http://gpars.org/guide/guide/5.%20Actors.html