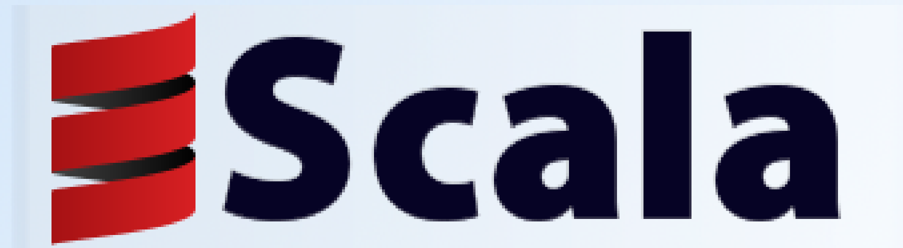


for Jarheads

Brian Clapper, *bmc@ardentex.com*
Mark Chadwick, *mark.chadwick@gmail.com*

Purpose of this presentation

We plan to introduce you to Scala, a language on the Java VM.

We hope to convince you that it solves many of Java's problems, while introducing some useful new features.

Purpose of this presentation

This presentation is *not* a comprehensive overview of Scala.

It's not even a comprehensive comparison of Scala and Java.

Still, we think it'll whet your whistle.

Purpose of this presentation

We'll try not to bore you.

Two Parts

There are two parts to this presentation:

- An introduction to Scala, for people coming from Java (Brian Clapper)
- A discussion of an amazingly cool Scala application, with hard performance data (Mark Chadwick)

Part 1: What is Scala?

What is Scala?

- Scala is a general purpose programming language that compiles down to JVM byte code. Just like Java.
- Scala is *not* a scripting language; it is compile-time type-safe. Just like Java. (You *can* script with it, though.)
- Scala is more concise and more powerful than Java.
- Scala is interoperable with Java.

What is Scala?

Scala has features Java lacks, including:

- functional *and* object-oriented support
- type inference, allowing simpler syntax without sacrificing compile-time type safety
- closures
- “type-safe duck typing”
- the ability to extend third-party APIs with programmable type conversion
- an actor-based concurrency library
- powerful pattern matching
- language-level support for XML
- and much more

Let's start with functions

Scala, like Java, has functions. They look a little different though.

Functions are important in Scala, so it's useful to spend a few minutes introducing them. Plus, this will introduce a few Scala concepts.

Let's start with functions

Here's a function to compute the sum of two numbers:

```
def sum(a: Int, b: Int): Int = {
    a + b
}
```

Note the lack of a `return` statement. In most cases, they're optional.

Let's start with functions

Since the function has only one statement, it can be simplified:

```
def sum(a: Int, b: Int): Int = a + b
```

But we're not done yet...

Let's start with functions

Since **a** and **b** both have type `Int`, the Scala compiler can infer that the return value is `Int`, so we can leave the return type off the declaration:

```
def sum(a: Int, b: Int) = a + b
```

(We'll talk more about type inference a little later.)

Let's start with functions

Notice the lack of semicolons. You can omit them, as long as the compiler isn't confused. For instance:

```
def factorial(n: Int): Int = {
    if (n == 0)
        1
    else
        n * factorial(n – 1)
}
```

This code compiles just fine. (It runs fine, too.)

Let's start with functions

Scala also does tail-call optimization (TCO), provided the recursive call is the *last* operation in the code.

The `factorial` function isn't tail-recursive, because the multiplication happens *after* the recursive call:

```
def factorial(n: Int): Int = {
    if (n == 0)
        1
    else
        n * factorial(n – 1)
}
```

Let's start with functions

But there is a way to write a TCO version of the `factorial` function. (This version is courtesy Daniel Spiewak.)

```
def factorial(n: Int) = {
    def doFactorial(n: Int, acc: Int): Int =
        if (n <= 0)
            acc
        else
            doFactorial(n - 1, acc * n)

    doFactorial(n, 1)
}
```

Because the recursive call is the last thing in the `doFactorial` function, Scala optimizes it into a loop.

First class functions

Scala has *first-class functions*.

This means you can also write functions as unnamed literals and pass them around as values.

(This is one reason I keep saying “function”, rather than “method”.)

Here's our **sum** function as a function literal, stored in a value:

```
val sum = (a: Int, b: Int) => a + b
```

First class functions

Function literals can be passed around. For instance, consider this (rather pointless) function:

```
def doIt(a: Int, b: Int,
         op: (Int, Int) => Int) = op(a, b)
```

`doIt()` takes two `Int` parameters, **a** and **b**, and an operation to perform on those parameters. It returns the result of the operation.

First class functions

Here's the function again, so we don't forget it:

```
def doIt(a: Int, b: Int,
         op: (Int, Int) => Int) = op(a, b)
```

`op` can be our `sum()` function from earlier. It can also be a function literal, composed on the fly.

```
doIt(10, 100, sum)
doIt(5, 50, (x, y) => x * y)
```

It can also be a member function of an object.

val, var and def

- Three ways to define class or object fields:
  - val
  - var
  - def
- `val` defines an immutable value (like a final variable)
- `var` defines a mutable value (like a "regular" variable)
- `def` defines a function
- For instance:

```
val x = 2 + 3  // x is 5, and constant
var x = 2 + 3  // x is 5, for now
def x = 2 + 3  // x computes 2 + 3 every time
```

Java irritants

Having introduced Scala functions (as well as `var` and `val`), we can now start to talk about how Scala addresses some of Java's shortcomings.

The problem with interfaces

- Interfaces can be very useful.
- But they're overused in Java.
- Or, more to the point, they're used where other constructs would be better.
- That is, if Java had other constructs.
- (Which it doesn't.)

Callbacks

Interfaces make lousy callbacks.

Interfaces as callbacks

In Java, we often use interfaces for callbacks.

- You define an API that uses callbacks (e.g., an event API).
- You specify that callbacks must implement some interface.
- Callers implement that interface so they can be called back.
- Swing uses this pattern heavily.

Interfaces as callbacks

This pattern has some serious drawbacks.

- You have to implement an entire interface, even if you only want a single function to be called back.
- Abstract adapters help, but only for one interface. What if you want one object to be called back via multiple APIs?
- Anonymous inner classes can help—but they cannot close over non-final variables.

Interfaces as callbacks

- These drawbacks drive dynamic language enthusiasts *crazy.* (They drive me crazy, too.)
- In a dynamic language, like Python, you can pass functions around.
- Those functions can live anywhere—in an object, by themselves, nested in another function.
- But Python isn't type-safe.
  - If you pass a bad function as a callback, you won't find out until runtime.
  - Which sucks.
  - Especially if “runtime” is halfway through a job that takes five hours to execute.

Callbacks in Scala

In Scala, you have more flexibility, as with Python or Ruby, but it's still type-safe.

First, let's assume an event-handling package with the following API:

```
def registerEvent(e: Event,
                  cb: Event => Unit)
def waitForEvents()
```

Let's also assume the existence of some event constants, like `Event.KeyPress`

Callbacks in Scala

In this function definition:

```
def registerEvent(e: Event,
                  cb: Event => Unit)
```

the type of **cb** is “function taking an `Event` parameter and returning nothing”.

`Unit` is like `void`.

(Don't get too hung up on the syntax for now.)

Callbacks in Scala: Use a member function

One obvious way to specify the callback is with a class (member) function:

```
class MyEventThing {
    def handleKeyPress(e: Event) =
        println(source + “ pressed a key“)

    def run() {
        event.registerEvent(Event.KeyPress,
                            handleKeyPress)
        event.waitForEvents()
    }
}
```

Callbacks in Scala: Use a nested function

Another way is with a nested function:

```
def waitForKeyPress(source: String) {

    def callback(e: Event) =
        println(source + “ pressed a key“)

    event.registerEvent(Event.KeyPress,
                        callback)
    event.waitForEvents()
}
```

Callbacks in Scala: Use a function literal

Yet another way is with a function literal:

```
def waitForKeyPress(source: String) {

    event.registerEvent(Event.KeyPress,
                             e => {println(“Got “ + e)})
    event.waitForEvents()
}
```

Callbacks in Scala: Structural types

- *Structural types* are like type-safe duck typing
  - If it walks like a duck and quacks like a duck, call it a duck.
- A structural type allows you to specify a type by specifying the *characteristics* of the type
- The simplest example is a function characteristic
- We want to specify the `registerEvent` method so that it accepts *any* object that has a `handleEvent` method...
- ... *no matter what that object's parent classes are.*

Callbacks in Scala: Structural types

Here's our new `registerEvent` signature:

```
def registerEvent(e: Event,
                  {def handleEvent(e: Event)})
```

That method can be called with *any* object that has a matching `handleEvent` method.

This is akin to duck typing, except that it's type safe.

Callbacks in Scala

- With the nested function and the function literal, the callback closes over any local variables.
- It closes over the *variables*, not their values. If a variable's value changes, the function picks up the new value.
  - This is *very* different from Java's inner classes, which can only close over a final value.

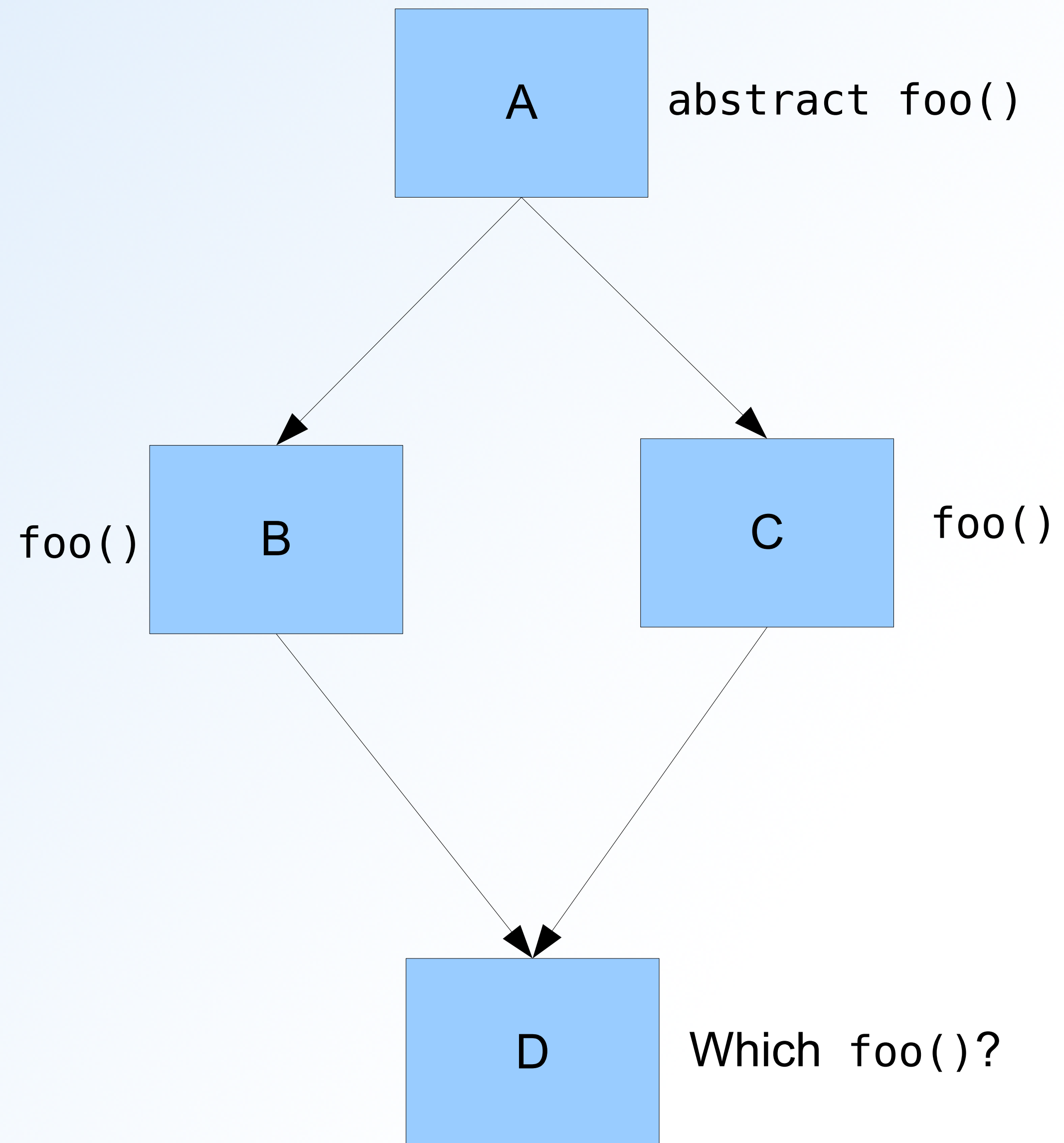
Interfaces and Inheritance

Ever want to put code in an interface?

I have.

Interfaces can't have code

- In Java, a class can have only one superclass, but can extend many interfaces.
- But—interfaces cannot contain executable code.
- With this limitation enforced, Java avoids the diamond inheritance problem.



Interfaces can't have code

But there are times when you'd really like to put common code in an interface. For instance:

- Lots of classes implement the interface, but ...
- ... it's impossible to create a common base class.

Scala traits: Interfaces with more vitamins

Scala has *traits*, which are like interfaces, except:

- Traits can contain executable code
- They can be *stacked* (more on this later)
- Conceptually similar to Ruby's module mixins

Scala traits: Interfaces with more vitamins

Before going further, let's look at an example (stolen from *Programming in Scala*):

```
trait Philosophical {
    def bloviate() =
        println("I use RAM, therefore I am.")
}
```

This trait can now be *mixed in* to a class:

```
class Frog extends Philosophical { ... }
val frog = new Frog

frog.bloviate() // prints "I use RAM ..."
```

Scala traits: Interfaces with more vitamins

To mix a trait into a class with a superclass, use *extends* for the class and *with* for the trait:

```
class Animal { ... }
class Frog extends Animal with Philosophical
```

You can mix in multiple traits:

```
trait HasLegs { ... }

class Frog extends Animal
           with HasLegs
           with Philosophical
```

Scala traits: Interfaces with more vitamins

Traits are sort of like interfaces *and* classes.

- Traits can declare fields and have state (like classes)
- Classes can mix in more than one trait (like interfaces)

One big difference:

- With classes, *super* is statically bound. That is, a class's superclass is known when the class is defined.
- With traits, *super* is dynamically bound. It's not known until the trait is mixed into some class.
  - This feature allows traits to have code while avoiding the diamond inheritance problem.
  - It also permits stackability.

Scala traits are stackable

Simple example:

```
class IntQueue {
    def get(): Int { ... }
    def put(x: Int) { ... }
}
trait Doubling extends IntQueue {
    // “extends” says it only works when mixed
    // into “IntQueue” classes
    abstract override def put(x: Int) {
        super.put(2 * x)
    }
}
trait Negating extends IntQueue {
    abstract override def put(x: Int) {
        super.put(-x)
    }
}
```

Scala traits are stackable

Now, we can create a new `IntQueue` that uses the traits:

```
val q = new IntQueue with Doubling

q.put(10)
println(q.get) // prints 20

val q2 = new IntQueue with Doubling with Negating

q2.put(10)
println(q2.get) // prints -20
```

The traits stack on top of one another, so a call to `put()` calls the top-most trait's `put()`, which in turn invokes the next one down the stack, and so on.

In this way, traits can be used to augment class behavior.

Moving past interfaces...

Verbosity

- Java is often verbose
  - Not as verbose as COBOL, but still...
- Languages like Python and Ruby are less verbose, but still readable
- Scala is less verbose than Java, too
  - Type inference has a lot to do with that
  - You can often omit parentheses, semicolons, and periods
- With Scala, code can be more terse, while still being readable.
  - Less to type
  - Less to wade through when reading

Verbosity & Type Inference

- Scala can often infer types, so you don't have to specify them. Examples:

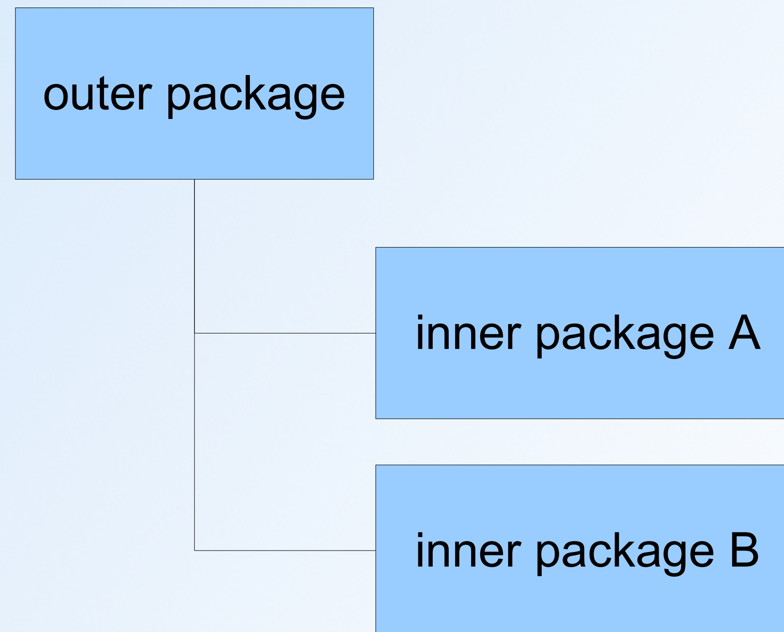
```
// x is obviously an Int. s is obviously a String.
// So is s2.
val x = 100
val s = “Foo”
val s2 = x.toString

// add() clearly returns an Int
def add(i: Int, j: Int) = i + j

// Scala will infer types for overridden methods
class Foo {
    override def toString = “foo”
}
```

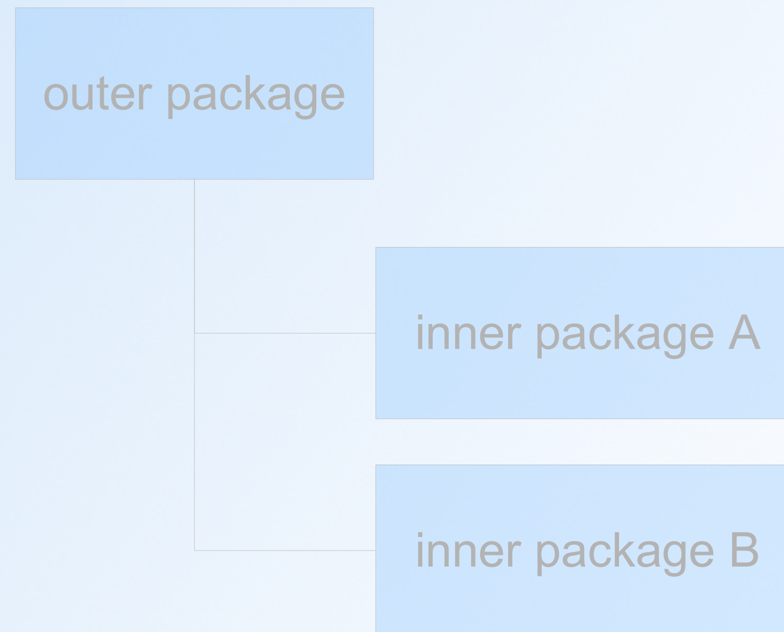
Imports, packages, and access modifiers

How many times have you had a class hierarchy like this?



Imports, packages, and access modifiers

The outer package has classes with internal methods that you want to share with hidden classes in the inner packages.



But, to do that, you have to make those common, internal methods *public*, because that's the only way to make them visible to the inner packages' classes.

Imports, packages, and access modifiers

- Java has no way to express that organization.
- Scala does.
- Scala's imports and packages are, in general, much more flexible than Java's.

Imports, packages, and access modifiers

First, Scala's packages can be nested within the code:

```
package OuterPackage {
    class Foo { ... }

    package InnerPackageA {
        class Bar { ... }
    }
}
```

The inner package's fully-qualified name is `OuterPackage.InnerPackageA`.

Unlike Java, `InnerPackageA` is truly, semantically *inside* `OuterPackage`.

Imports, packages, and access modifiers

There *is* a Java-like shorthand, for simple packages that live in a single file:

```
package SimplePackage

class Foo { ... }
```

Imports, packages, and access modifiers

Imports are also slightly different. Examples are easiest here:

```
import foo.Bar              // “Bar” class now in scope
import foo.{Bar => Baz}     // “Bar” in scope, as “Baz”
import foo._                // like Java's “import foo.*”
```

Imports can also be nested inside code:

```
def someFunction() {
    import foo.Bar
    val b = new Bar()
    ...
}
```

Imports, packages, and access modifiers

- As in Java, members of packages and classes (and Scala objects) can be labeled with access modifiers.
  - **Private** members are visible only inside the class or object that contains the definition.
  - **Protected** is more restrictive than Java: Only subclasses can access protected members.
  - There is no package-visible default access modifier. Members with no access modifier are public.

Imports, packages, and access modifiers

Here's where it gets interesting.

- Access modifiers can be further scoped with *qualifiers* of the form `private[X]` or `protected[X]`.
- The “X” part means “up to X”, where “X” is some enclosing package, class or singleton object.
- `private[this]` means “private to the current *object*”
- Qualifiers give a finer-grained control over access modifiers.

Imports, packages, and access modifiers

```
package outer {
    package inner1 {
        // Foo is visible to everything in
        // package “outer”, private elsewhere.
        private[outer] class Foo { ... }
    }
    package inner2 {
        // Foo is visible here.
        class Bar extends Foo { ... }
    }
}
```

You can easily craft packages that carefully share members among themselves, while hiding them from everyone else.

Setters and getters

- In Scala, setters and getters are more like Python properties.
- That is, callers can access members as if they were fields, but they're really invoking hidden setters and getters.
- You can override the default setters and getters to alter access behavior...
- ... without affecting the callers.

Setters and getters

For example. This:

```
class Person(personName: String) {
    var name: personName
}
```

is the same as:

```
class Person(personName: String) {
    private[this] var n: personName

    def name: String = n              // getter
    def name_=(s: String) {n = s}  // setter
}
```

You can redefine the setter and getter methods. The caller still gets to access the value with field syntax.

Setters and getters

- If you need to make a class use Java-style setters and getters (e.g., to interact with Spring), you can do this:

```
class Person(personName: String) {
    @BeanProperty
    var name: personName
}
```

- Scala will generate bean-style `setName()` and `getName()` methods.

Operator Overloading

- Scala lets you overload operators
- Operators are just methods
- Highly useful for internal Domain Specific Languages (DSLs)
- Imagine a Scala-based build language (a Scala analog to Rake), with syntax like this:

```
“foo.exe” -> (“foo.o”, “bar.o”) := {
    cc(“-o”, “foo.o”, “bar.o”, OtherLibs)
}

“foo.o” -> “foo.c”
```

- With operator overloading (and some other stuff), this is legal Scala

Super "for" loop

- Scala's *for* loop is really powerful
- It can do the usual stuff:

```
for (i <- 1 to 10) println("Iteration " + i)

for (file <- new java.io.File(".").listFiles)
    println(file)
```

- It can also do more complicated things, such as guard conditions:

```
for (file <- new java.io.File(".").listFiles
     if (! file.getName.startsWith(".")))
    println(file)
```

Super “for” loop

- ... and nested loops:

```
import scala.io.Source
import java.io.File

def getLines(file: File) =
    Source.fromFile(file).getLines.toList

def grep(regexp: String) =
    for {
        file <- new File(“.”).listFiles
        if file.getName.endsWith(“.scala”)
        line <- getLines(file)
        trimmed = line.trim
        if (trimmed.matches(regexp))
        }
        println(file + “: “ + trimmed)
```

Super “for” loop

- ... and generators (that are similar to Python list comprehensions):

```
import scala.io.Source
import java.io.File

for {
    file <- new File(“.”).listFiles
    name = file.getName
    if (name.endsWith(“.scala”))
    (line, n) <- getLines(file).zipWithIndex
    trimmed = line.trim
    if (trimmed.matches(regexp))
    }
    yield name + “:” + (i+1) + “ “ + line
```

More fun with closures

- We've all had to write code like this:

```
lock.acquire();
try {
    performLockedOperation();
}
finally {
    lock.release();
}
```

- We've also had to debug code where the `finally` clause is omitted, resulting in deadlock.

More fun with closures

- What a mess.
- Wouldn't it be better to have the locking API *provide* a method that asserts the lock and executes your code for you?
- In Java, you can use callbacks that way.
- Sometimes you can use annotations.
  - `@Transactional`, for instance
- But it's much, much cleaner with closures.

More fun with closures

With function literals (and a little syntactic sugar), Scala lets you do this:

```
def withLock(lock: Lock)(code: => Unit) {
    lock.acquire();
    try {
        code
    }
    finally {
        lock.release();
    }
}
```

You can then call it like this:

```
withLock(lock) {
    code
}
```

Pattern matching

- Scala has built-in general pattern-matching
- It's *really* powerful, and it's all over the language and all through its libraries

Pattern matching

- Here's an example that matches integer values:

```
def testMatch(x: Int): String =
    x match {
        case 0 => "none"
        case 1 => "one"
        case 2 => "two"
        case _ => "many"
    }
```

- The `testMatch()` function maps integers to strings.

Pattern matching

- Here's another example, matching against patterns of different types:

```
def testMatch(x: Any): Any =
    x match {
        case 0      => “none”
        case “one”  => 1
        case y: Int => “scala.Int=” + y
    }
```

- Used like that, it's `switch` on steroids.

Pattern matching

- Pattern matching can be used in a variety of ways
- Here's a pattern match that discards the first element of a list:

```
def dropl[T](l: List[T]): List[T] =
    l match {
        case Nil => Nil
        case item :: sublist => sublist
    }
```

- Pattern matching can get even more complicated than that.

Exceptions

- Exceptions in Scala are thrown the same way as in Java
- But they're caught a little differently. Note the pattern matching:

```
import java.io._

try {
    val f = new FileReader("foobar.txt")
}
catch {
    case ex: FileNotFoundException => ...
    case ex: IOException => ...
}
```

- You also don't have to declare or catch checked exceptions.

No static methods

- There are no static methods in Scala
- Instead, you use special singleton objects
- These objects can also be paired with classes (so-called *companion* objects), allowing them to do special things
- These singleton objects can encapsulate common methods
- When imported, they look rather like Python-style modules

Concurrency

- One of Java's strengths is concurrency

Concurrency

- One of Java's strengths is concurrency
- But it's also a weakness

Concurrency

- One of Java's strengths is concurrency
- But it's also a weakness
- Threads don't scale

Concurrency

- One of Java's strengths is concurrency
- But it's also a weakness
- Threads don't scale
- `java.util.concurrent` is excellent, but it's also easy to get wrong

Concurrency

- One of Java's strengths is concurrency
- But it's also a weakness
- Threads don't scale
- `java.util.concurrent` is excellent, but it's also easy to get wrong
  - The API patterns can be complicated
  - You can screw up locking
  - etc.

Concurrency with Actors

- Actors are not new to Scala (c.f., Erlang)
- There are Java implementations, but the Scala implementation has nicer syntactic sugar
- An Actor is a thread-like object that has a mailbox for receiving messages
- The Actor library handles the concurrency issues. All you do is pass messages.
- Two basic patterns:
  - One Actor per thread (easier to write; doesn't scale)
  - Actors multiplexed across threads (a wee bit harder to write, but scales better)

Concurrency with Actors

- A super-simple example: An echo client and server.
- Here's the server:

```
class EchoServer extends Actor {
    def act() {
        while (true) {
            receive {
                case (s: String, sender: Actor) =>
                    println("Server: Got: " + s)
                    sender ! s
                case "EXIT" =>
                    println("Server: Exiting.")
                    exit()
            }
        }
    }
}
```

Concurrency with Actors

- Here's the client:

```
import scala.actors.Actor

class EchoClient(val server: Actor) extends Actor {
    def act() {
        for (i <- 1 to 5) {
            val msg = “Message” + i
            println(“Sending: “ + msg)
            server ! (msg, this)
            receive {
                case (s: String) =>
                    println(“Got back: “ + s)
            }
        }
        server ! “EXIT”
    }
}
```

Concurrency with Actors

- Here's the test runner:

```
import scala.Application

object tester extends Application
{
    val server = new EchoServer()
    val client = new EchoClient(server)
    server.start()
    client.start()
}
```

Concurrency with Actors

- Compile the test runner and run it, and here's the output:

```
$ scala -cp . tester
Client: Sending: Message 1
Server: Got: Message 1
Client: Got back: Message 1
Client: Sending: Message 2
Server: Got: Message 2
Client: Got back: Message 2
Client: Sending: Message 3
Server: Got: Message 3
Client: Got back: Message 3
Client: Sending: Message 4
Server: Got: Message 4
Client: Got back: Message 4
Client: Sending: Message 5
Server: Got: Message 5
Client: Got back: Message 5
Server: Exiting.
```

Concurrency with Actors

- Obviously, you can get a *lot* more complicated than that.
- Imagine, for instance, that Scala build tool.
  - Each target is an Actor
  - Each actor waits for "done" messages from all its dependent targets before building itself
  - Each target sends a message to a dispatcher when it's done building
  - The dispatcher sends the "done" message to all targets that depend on the one that just finished.
- A very simple model for a parallel build algorithm
- (I actually built that.)

Implicit conversions

- Scala permits scoped, implicit type conversions.
- They're useful for DSLs and extending final classes.
- For example, suppose you want to extend the `String` class to support this method:

```
public String delete(String substring)
```

- You can't do that in Java, because `String` is final.
- You can create your own `MyString` wrapper class that *looks* like `String`.
- But you must write wrapper methods for *every* `String` method.
- And you *still* have to do the conversion yourself.

Implicit conversions

- In Scala, create your `MyString` class, with only the new methods you need.
- Then create an implicit conversion function:

```
implicit def stringToMyString(s: String) =
    new RichString(s)
```

- If that function is in scope, you can do this:

```
val s = “foo bar baz foo”
val s2 = s.delete(“ baz”) // “foo bar foo”
```

- Scala finds no `delete()` in `String`, so it looks for an implicit conversion to something with `delete()`.
- It finds one, so it implicitly converts `String` to `MyString`, and calls `MyString`'s `delete()` method.

Functional constructs

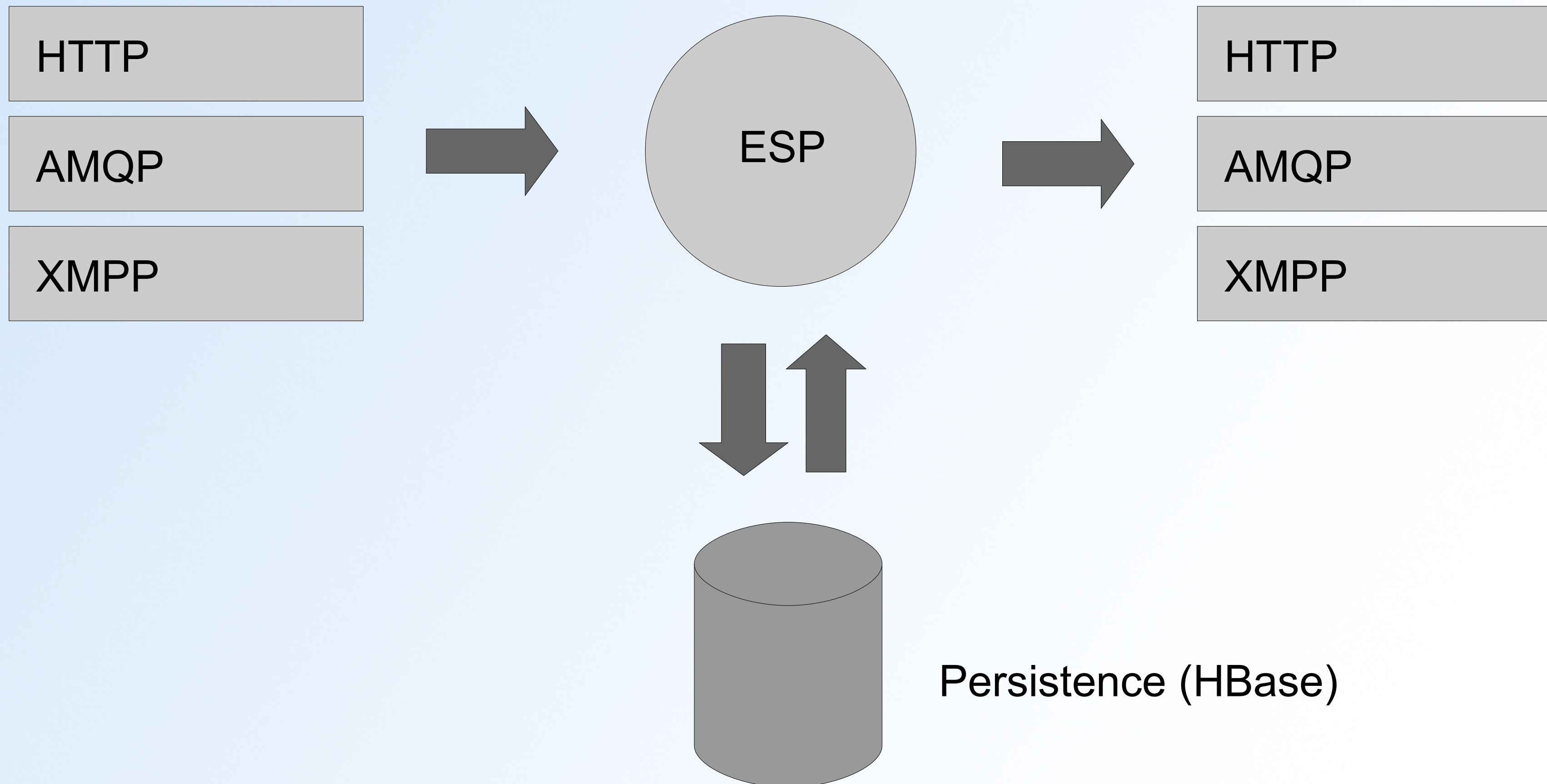
Without diving too deeply into functional programming here, Scala does have functional constructs, including:

- Immutable references and values (via `val`)
- Immutable collections (e.g., immutable Maps, Sets and Lists), which are the default.
- Nearly everything returns a value
  - even *for* loops and *if* statements

Part 2: A real-world Scala application

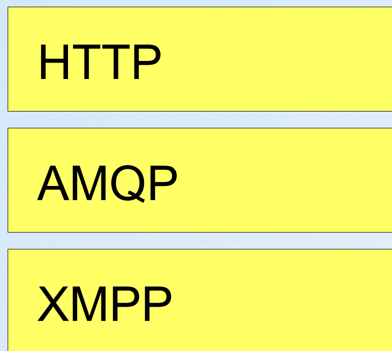
Building a "Complex Event Stream Processor" from the ground up with Scala.

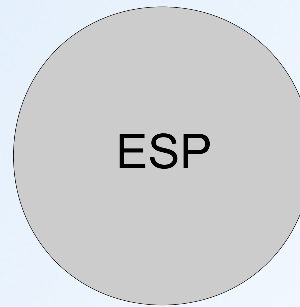
Event Stream Processors

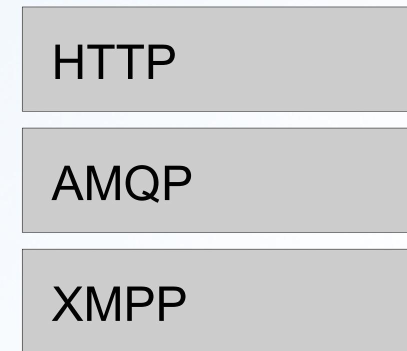


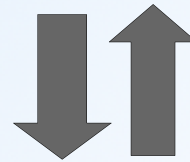
Event Stream Processors

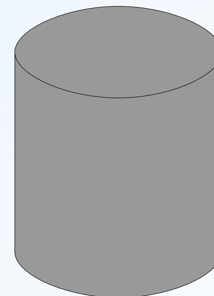
Low Latency





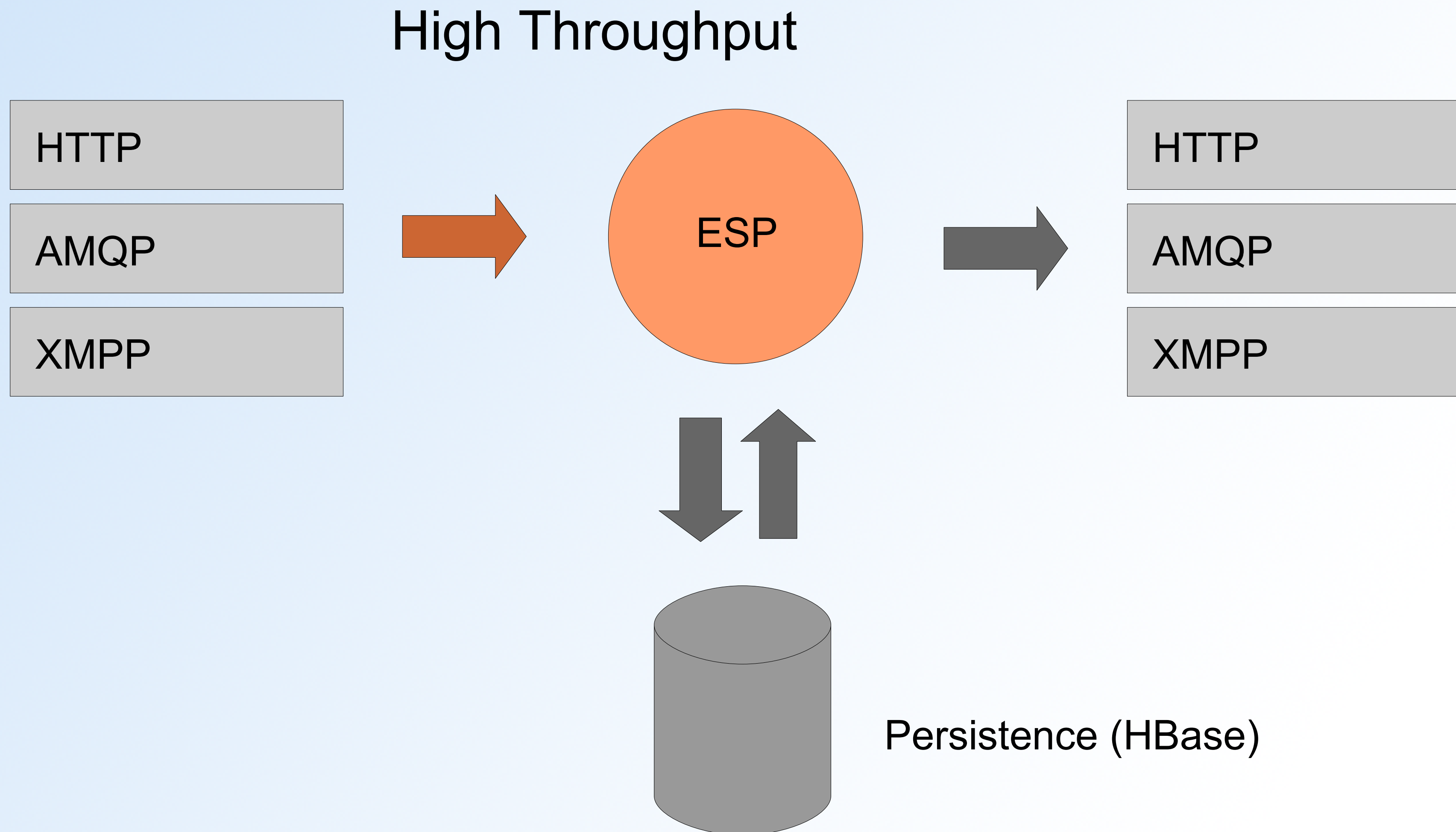




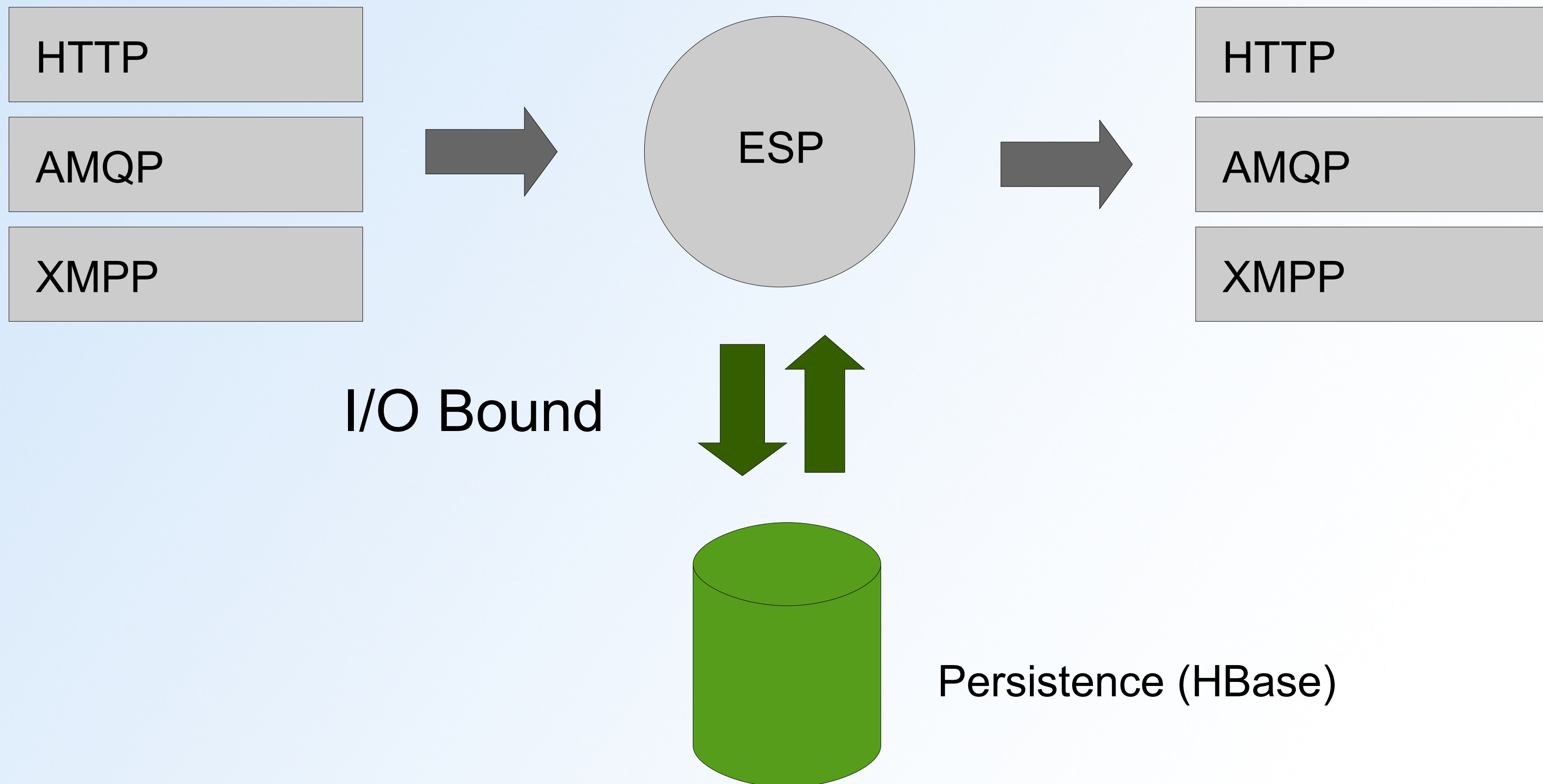


Persistence (HBase)

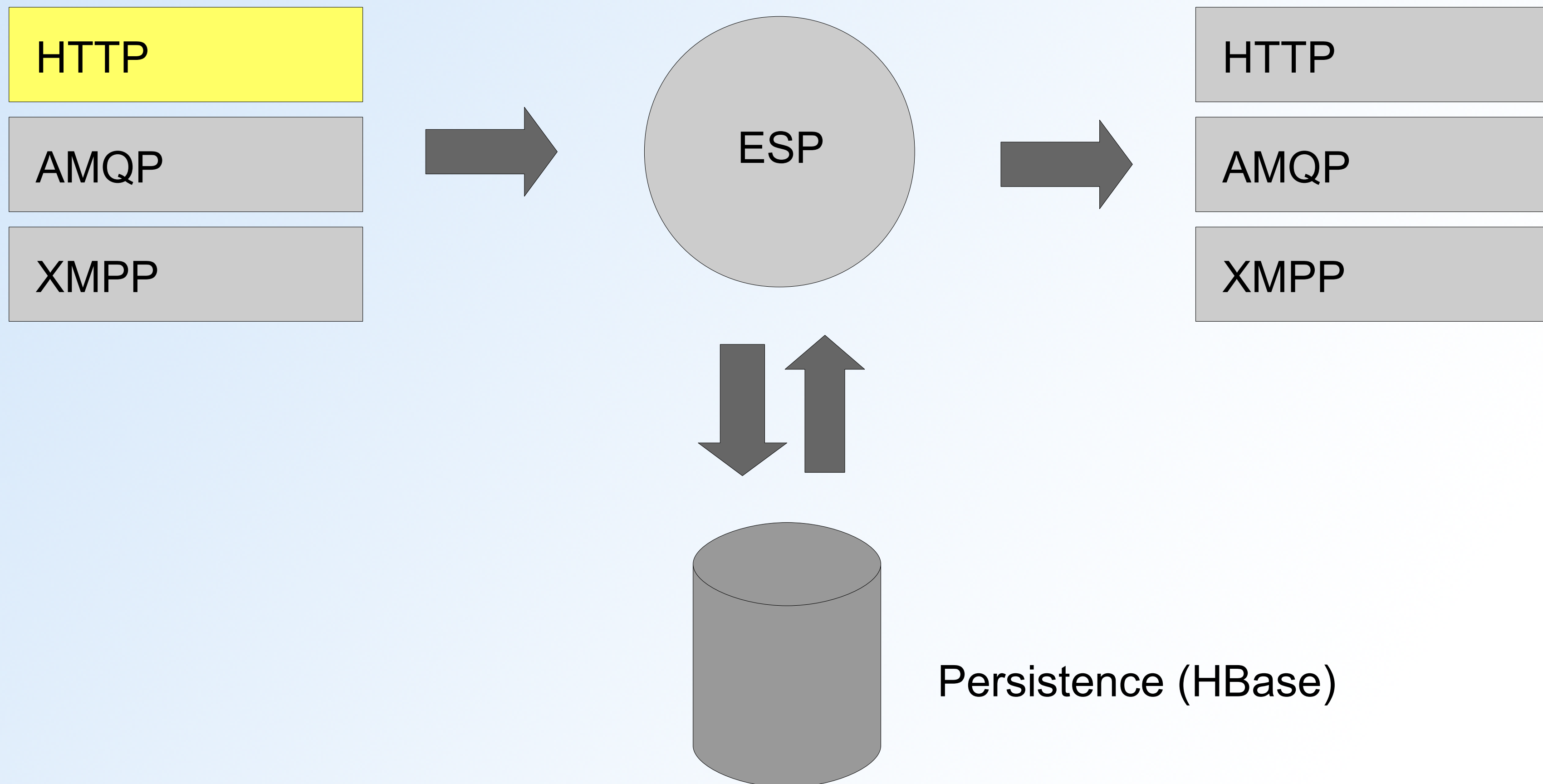
Event Stream Processors



Event Stream Processors



The HTTP Event Collector



The HTTP Event Collector

- Accept connection
- Parse request, determine validity
- Find routing information and matching callbacks
- Notify callbacks of matched event
- Persist message
- If not given, set cookie to uniquely identify user
- Close connection

The HTTP Event Collector

- 10,000 Requests
- 50 Concurrent
- Threadpool connector
- 4 Available threads
- Latency: 500ms
- Throughput: 12 reqs/sec

The HTTP Event Collector

- Accept connection
- **“Fire and forget” message to event processing pool**
- Parse request, determine validity
- Find routing information and matching callbacks
- Notify callbacks of matched event
- Persist message
- If not given, set cookie to uniquely identify user
- Close connection

The HTTP Event Collector

- 10,000 Requests
- 50 Concurrent
- APR connector
- Event-based (single thread)
- Latency: 1ms
- Throughput: 15000 reqs/sec

Big Wins

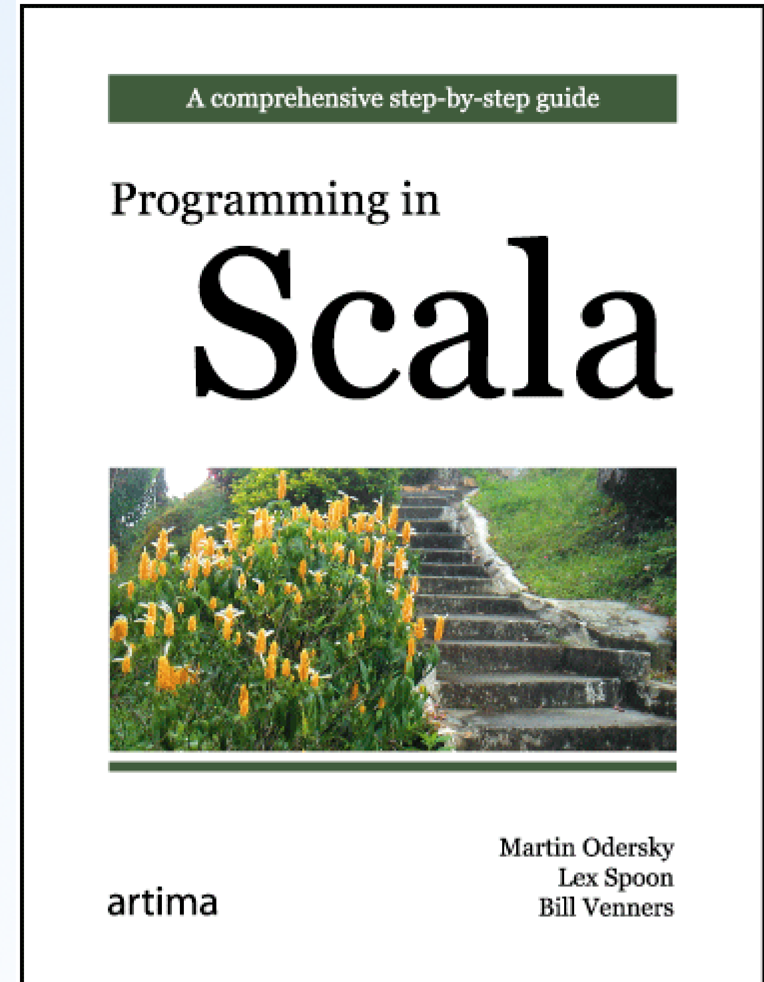
- Easy to reason about concurrent and asynchronous actions
- Huge selection of libraries to do my heavy lifting
- Builds a war file. Easy to deploy to any servlet container.
- Terse and intuitive code thanks to inferred types, traits and blocks/closures.
- Very comprehensive tests thanks to Scalacheck

Lessons Learned

- Push actors as low as possible (but no lower). Have them do as little as possible.
- Case classes may lead to coupled code. Use them sparingly, and “seal” them when logical.
- IDE support is going to be lacking for a while. Get a build system early.
- Message passing is not free.
- The cool syntactic sugar sprinkled around while learning Scala is not so cool weeks later.

More information about Scala

- Start with *Programming in Scala*. It's both an excellent reference and a superb overview. See www.artima.com
- The Scala web site, www.scala-lang.org
- The #scala IRC channel (on irc.freenode.net)
- The Scala mailing list (see the web site)



Acknowledgments

Thanks to Toby DiPasquale, Paul LaFollette and Steve Sapovits, for reviewing this presentation.

The last slide. (Really.)

Questions?
Comments?
Rotten tomatoes?