# Making Java APIs usable with Scala

Josh Cough
joshcough@gmail.com

# Who Am I?

## VP of Tremendous Power at S&P Capital IQ

- Ermine, Haskell, Scala, F#, JavaScript, C#, ~~Java~~

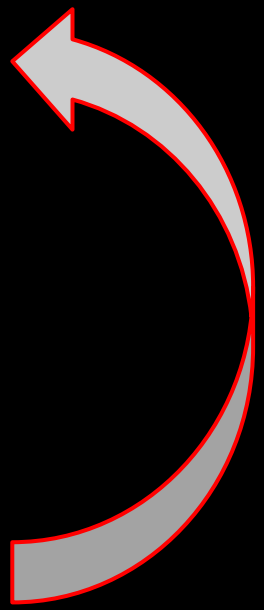
## Open Source

- https://github.com/joshcough/
- https://github.com/joshcough/MinecraftPlugins
- ScalaTest, SBT (a little), Scalaz (a tiny bit)

# Not Me:

# The Plan

1. What is Minecraft? What is Bukkit?
2. Some Bukkit examples in Java
3. Same code redone with Scala
4. Then lots of explaining how



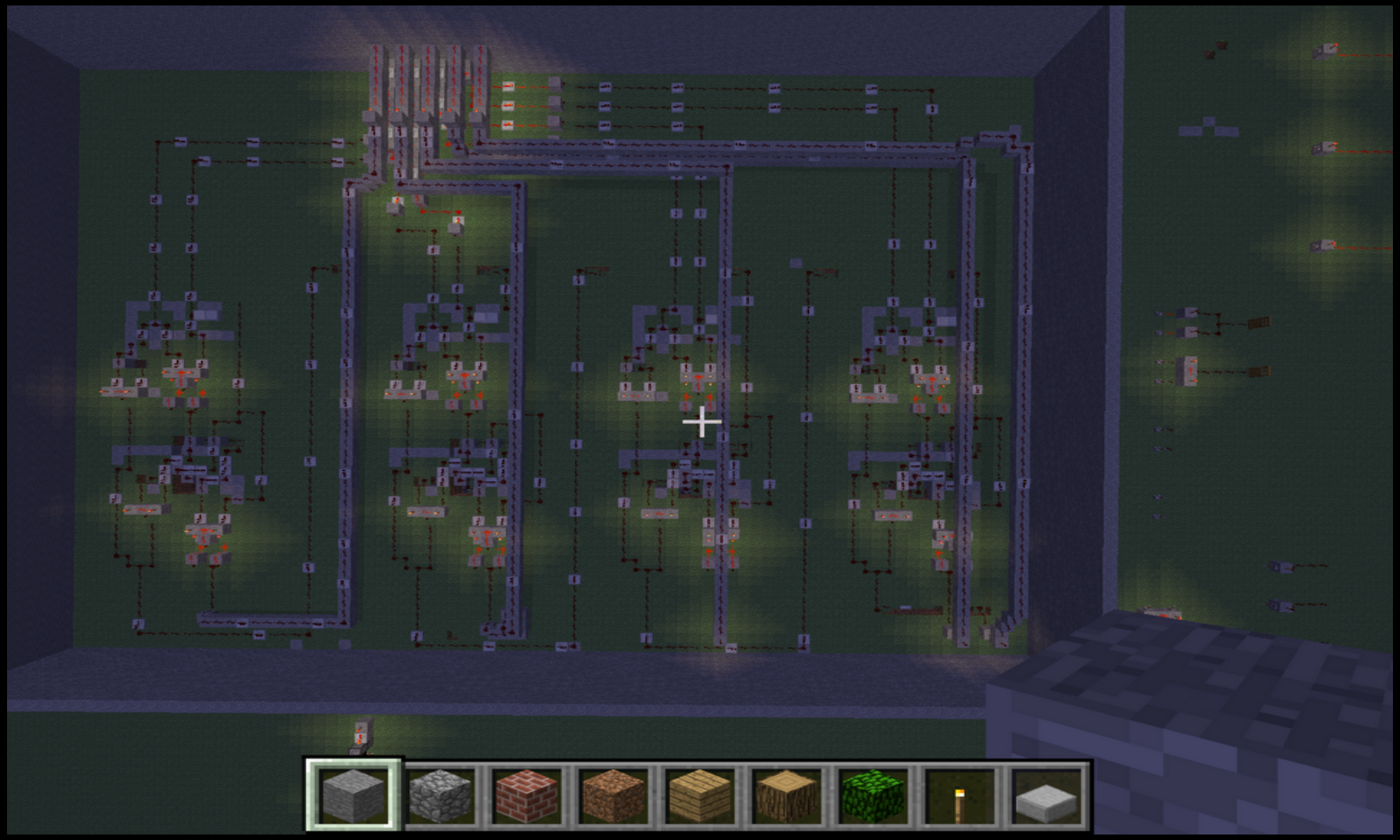
# What is Minecraft?

- An awesome game
- An outlet for creativity
- A real world Java project
  - Over 40 million registered users
  - $200m+ in sales.
- An opportunity to get people coding

# Awesomeness

# Creativity

# Creativity



# What is Bukkit?

## Minecraft Server Plugin API (bukkit.org)

- Event Listeners

  An API to respond to all sorts of events that happen on the server.

- Commands

  An API to handle user commands.

- Some more stuff, but out of scope

# Bukkit Listeners

```
// 1:  Extend JavaPlugin
public class BlockChangerGold extends JavaPlugin {

  // 2:  Create a handler for the event
  class BlockChangerListener implements Listener {
    @EventHandler
    public void onBlockDamage(BlockDamageEvent event){
      event.getBlock().setType(Material.GOLD_BLOCK);
    }
  }

  public void onEnable() {
    // 3:  Register the handler for the event
    registerEvents(new BlockChangerListener(), this);
  }
}
```

# Scala Time!

# Features

## Things we'll definitely cover

- Higher Order Functions
- "Enrichment Classes" (new in 2.10*)
- String Interpolation (also new in 2.10)
- Parser Combinators
- Maybe more, time permitting.

# Bukkit Listener in Scala

```
class BlockChangerGold extends ListeningFor(
  OnLeftClickBlock(
    (player, event) => event.block changeTo GOLD_BLOCK
))
```

# Bukkit Listener in Scala

```
class BlockChangerGold extends ListeningFor(
  OnLeftClickBlock((_, e) => e.block changeTo GOLD_BLOCK)
)
```

# Oh wait...that's better.

```
class BlockChangerGold extends ListeningFor(
  OnTouch((p, e) =>
    if (p is "joshcough") e.block changeTo GOLD_BLOCK
))
```

# How did we get here?

- Higher Order Functions
- Function Literals
- Enrichment Classes

# Higher Order Functions

Assertion #1: A Listener is just a Function.
Assertion #2: Functions are just Objects.

```
class MyListener implements Listener {
    public void onBlockDamage(BlockDamageEvent event)
}
```

```
class MyListener extends Function1<BlockDamageEvent, Void>{
    public void apply(BlockDamageEvent event)
}
```

# Function Literals

Assertion #3: Functions are easier to create.

```
// The hard way to create a Function object
new Function1[PlayerInteractEvent, Unit]{
  def apply(e:PlayerInteractEvent): Unit = println(e)
}
```

```
// The easy way
(e:PlayerInteractEvent) => println(e)
```

```
// And in the right context, an even easier way
e => println(e)
```

# HOFs vs Listeners: Quick Review

```
class BlockChangerGold extends ListeningFor(
  OnLeftClickBlock(
    (p, e) =>
      if (p is "joshcough") e.block changeTo GOLD_BLOCK
  )
)
```

# HOFs vs Listeners: Quick Review

```
class BlockChangerGold extends ListeningFor(
  OnLeftClickBlock(
    (p, e) =>
      if (p is "joshcough") e.block changeTo GOLD_BLOCK
  )
)
```

```
def OnLeftClickBlock(f: (Player, PlayerInteractEvent) => Unit) =
  new Listener {
    @EventHandler
    def on(e:PlayerInteractEvent) =
      if (e.getAction == LEFT_CLICK_BLOCK) f(e.getPlayer, e)
  }
```

# HOFs vs Listeners: Quick Review

```
class BlockChangerGold extends ListeningFor(
  OnLeftClickBlock(
    (p, e) =>
      if (p is "joshcough") e.block changeTo GOLD_BLOCK
  )
)
```

# HOFs vs Listeners: Quick Review

```
class BlockChangerGold extends ListeningFor(
  OnLeftClickBlock(
    (p, e) =>
      if (p is "joshcough") e.block changeTo GOLD_BLOCK
  )
)
```

```
(Player, PlayerInteractEvent) => Unit
```

```
Function2[Player, PlayerInteractEvent, Unit]
```

# Enrichment Classes (Scala 2.10*)

```
implicit class RichPlayer(player:Player){
  def is(name: String) = player.getName == name
  def ! (s:String) = if(s != null) player.sendMessage(s)
  def shock = world strikeLightning player.getLocaction
}
```

```
if(player is "joshcough"){
  player.shock
  player ! "zap!"
}
```

# Enrichment Classes (Scala 2.10*)

```
implicit class RichBlock(b:Block) {
  def changeTo(m:Material) = b setType m
  def isNot    (m:Material) = b.getType != m
}
```

```
implicit class RichPlayerInteractEvent(e:PlayerInteractEvent){
  def block = e.getClickedBlock
}
```

```
if(e.getClickedBlock.getType != AIR)
  e.getClickedBlock setType STONE
```

```
if(e.block isNot AIR) e.block changeTo STONE
```

# String Interpolation (2.10)

```
// Simple variable inside
p ! s"bc using: $m"
```

```
// Expression inside
p ! s"${p.name}, bc using: $m"
```

```
// Nested is ok too
p ! s"Awesome! ${p.name + s", bc using: $m"}"
```

```
// Escaping as you'd expect (and required here).
p ! s"${p.name}, you have \$500"
```

# Commands

Let's change BlockChangerGold to allow:

- Changing the punching material
- Turning it off

Examples:

- `/bc stone`
- `/bc gold_block`
- `/bc` `*(this turns it off)`

# Quiz

What should happen if someone types these?

```
/bc gold_block

/bc

/bc eoriiweroijweorijwe

/bc dirt 7
```

# Quiz

What should happen if someone types these?

| | |
|---|---|
| `/bc gold_block` | **GOOD!** |
| `/bc` | **GOOD!** |
| `/bc eoriiweroijweorijwe` | **ERROR!** |
| `/bc dirt 7` | **ERROR!** |

# Bukkit Commands

```
// 1: still have to extend JavaPlugin
public class JavaPlugin {

    // 2: then implement this function
    //    to handle ALL of your commands
    public boolean onCommand(
        Player sender,
        String command,
        String[] args)
}
```

Disclaimer: This isn't exactly the API, but it's close enough for our purposes.

# BlockChanger Revisited

*Too big for slide!*

BlockChanger.java

# Problems with Commands

What's wrong with this API?

- A lot!
- And at least 37 other problems

# Parser Combinators

Assertion: Parser Combinators are the solution to *all* *42* of these problems.

Best to explain by example.

# Basic examples

```
run(int,  "5")          ⇒ Success(5,List())
run(bool, "true")       ⇒ Success(true,List())
run(anyString, "hello") ⇒ Success("hello",List())

run(int,  "x")          ⇒ Failure(invalid int: x)
run(bool, "x")          ⇒ Failure(invalid boolean: x)

run(eof, "")            ⇒ Success((),List())
run(eof, "blah")        ⇒ Failure(expected eof, but got: blah)


run(int, "7 8")         ⇒ Success(7,List(8))
```

# Combinator examples

```
run(int ~ int, "7 8")          ⇒ Success((7 ~ 8),List())
run(int ~ int, "5 x")          ⇒ Failure(invalid int: x)
run(int ~ anyString, "5 x")    ⇒ Success((5 ~ "qweqwe"),List())
run(int ~ bool, "5 true x")    ⇒ Success((5 ~ "true"),List(x))

run(bool or int, "true")       ⇒ Success(Left(true),List())
run(bool or int, "7")          ⇒ Success(Right(7),List())
run(bool or int, "qweqw")      ⇒ Failure(invalid boolean: qweqw or
                                         invalid int: qweqw)

run(bool ~> int, "true 8")     ⇒ Success(8,List())

run(bool <~ eof, "true")       ⇒ Success(true,List())
run(bool <~ eof, "true 8")     ⇒ Failure(expected eof, but got: 8)
```

# Lots of examples

```
// implicit conversion from string to Parser[String] here.
run("test", "test")    ⇒ Success("test",List())
run("test", "er")      ⇒ Failure(expected: test, but got: er)

run(int.? ~ "hi", "hi")    ⇒ Success((None ~ "hi"),List())
run(int.? ~ "hi", "6 hi") ⇒ Success((Some(6) ~ "hi"),List())

run(int.*, "5 7 8 9")      ⇒ Success(List(5, 7, 8, 9),List())
run(bool.+, "true false") ⇒ Success(List(true, false),List())

run(int ^^ (x => x * x), "7") ⇒ Success(49,List())
run(int ~ "*" ~ int ^^^ "hi!", "6 * 9") ⇒ Success("hi!",List())
```

# Minecraft Parsers

```
val gamemode: Parser[GameMode] =
  ("c" | "creative" | "1") ^^^ CREATIVE |
  ("s" | "survival" | "0") ^^^ SURVIVAL

val coordinates: Parser[Int ~ Int ~ Option[Int]] =
  int ~ int ~ int.?

// just the types for these to save time
val material: Parser[Material]
val player  : Parser[Player]
val location: Parser[World => Location]
val plugin  : Parser[Plugin]
val time    : Parser[Int] // but between 0 - 24000
```

# Bukkit Commands in Scala

```
class BlockChanger extends ListenerPlugin with CommandPlugin {
  val users    = collection.mutable.Map[Player, Material]()
  val listener = OnLeftClickBlock((p, e) =>
    for(m <- users.get(p)) e.block changeTo m
  )
  val command  = Command(
    name = "bc",
    desc = "Specify which material to change blocks to.",
    args = material or eof)(
    body = {
      case (p,Left(m)) => users+=(p -> m); p ! s"bc using: $m"
      case (p,_)       => users-=p;        p !  "bc disabled"
    }
  )
}
```

# Bukkit Commands in Scala

```
class BlockChanger extends ListenerPlugin with CommandPlugin {
  val users    = collection.mutable.Map[Player, Material]()
  val listener = OnLeftClickBlock((p, e) =>
    for(m <- users.get(p)) e.block changeTo m
  )
  val command  = Command(
    name = "bc",
    desc = "Specify which material to change blocks to.",
    args = material or eof)(
    body = {
      case (p,Left(m)) => users+=(p -> m); p ! s"bc using: $m"
      case (p,_)       => users-=p;        p !  "bc disabled"
    }
  )
}
```

# Quiz Revisited

## /bc gold_block

```
run(parser,"gold_block") ⇒ Success(Left(GOLD_BLOCK),List())
```

## /bc

```
run(parser, "")          ⇒ Success(Right(()),List())
```

## /bc x

```
run(parser, "x")         ⇒ Failure(invalid material-type: x or
                                   unprocessed input: x)
```

## /bc dirt 7

```
run(parser, "dirt 7")    ⇒ Success(Left(DIRT),List(7))
```

# Handle Extra Arguments

```
anyUserParser <~ eof
```

# Handle Extra Arguments

```
(material or eof) <~ eof
```

**/bc dirt 7**

```
run(material or eof, "dirt 7") ⇒
  Success(Left(DIRT),List(7))
```

```
run((material or eof) <~ eof, "dirt 7") ⇒
  Failure(expected eof, but got: 7)
```

# A few more Commands

```
Command("goto", "Teleport!", args = player or location) {
  case (you, Left(them)) => you.teleportTo(them)
  case (you, Right(loc)) => you.teleport(loc(you.world))
}
```

```
Command("set-time", "Sets the time.", args = time) {
  case (p, n) => p.world.setTime(n)
}
```

```
Command("gm", "Set your game mode.", args = gamemode) {
  case (p, gm) => p.setGameMode(gm)
}
```

# Putting it all together

WorldEditDemo.scala

And some for comprehensions too...