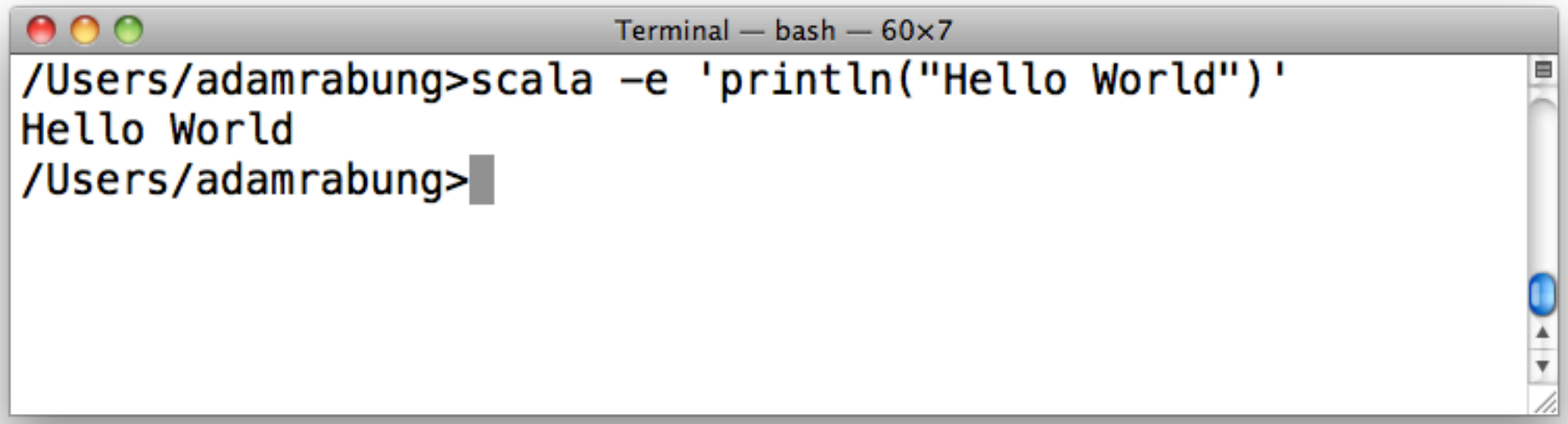
An Introduction to Scala for Java Programmers

Adam Rabung

In 2003, Martin Odersky and his team set out to create a "scalable" language that fused important concepts from functional and object-oriented programming. The language aimed to be highly interoperable with Java, have a rich type system, and to allow developers to express powerful concepts concisely.

Scalable Language



Scalable Language

Terminal — java — 79×16

```
/Users/adamrabung>scala
Welcome to Scala version 2.8.0.final (Java HotSpot(TM) 64-Bit Server VM, Java 1
.6.0_20).
Type in expressions to have them evaluated.
Type :help for more information.

scala> "cheeseburger".toUpperCase
res0: java.lang.String = CHEESEBURGER

scala> 8 + 19
res1: Int = 27

scala> Set(1,8,9).contains(8)
res2: Boolean = true

scala>
```

Simply Scala

Created by Anthony Bagwell

Home | Comments | About | Learn More Scala

```
Welcome to Simply Scala 2.8

      Scala is a modern computer programming language.
      Here you can discover more about its features in
      a simple interactive way.

Now available in French as well.

Creating user space...
Please wait until 'Ready for Code' message.
New interpreter instance being created for you, this may take a few seconds.
Ready for code.
println("hello world")
hello world
```

Reset

Evaluate

Scalable Language

```
object Hello {
    def main(args:Array[String]) {
        println("Hello World")
    }
}
```

```
/Users/adamrabung/hello>scalac Hello.scala
/Users/adamrabung/hello>ls
total 24
drwxr-xr-x   5 adamrabung  staff   170 Nov  7 20:32 .
drwxr-xr-x+ 42 adamrabung  staff  1428 Nov  7 20:32 ..
-rw-r--r--   1 adamrabung  staff   589 Nov  7 20:32 Hello$.class
-rw-r--r--   1 adamrabung  staff   609 Nov  7 20:32 Hello.class
-rw-r--r--   1 adamrabung  staff    81 Nov  7 20:31 Hello.scala
/Users/adamrabung/hello>java Hello
Hello World
```

Scalable Language

mention linkedin connect with jruby front and scala back

mention using clojure data structures in scala

Scala loves Java

```
import javax.servlet.http.HttpServlet
import javax.servlet.http.HttpServletRequest
import javax.servlet.http.HttpServletResponse

class ScalaServlet extends HttpServlet {

  override def doGet(request: HttpServletRequest,
                     response: HttpServletResponse) {

    response.getWriter().println("Hello World!")
  }

}
```

Java loves Scala

```
class CrazyScalaTechnology {
  def doWebThreeDotOhStuff(msg:String) {
    println(msg)
  }
}
```

← Scala

```
public class JavaPojo {
    public static void main(String[] args) {
        new CrazyScalaTechnology().doWebThreeDotOhStuff("Hello world!");
    }
}
```

← Java

eclipse supports mixed projects that can hold .scala and .java

note the String being passed in to Java - in fact, most of the scala "stdlib" is just javas

Highly interoperable with Java

- To your boss and the people who administer your servers, it's "just a jar"
- Because it's not an all or nothing proposition, you can slowly "sneak" Scala into your codebase - think test cases!

Scala is OO

- Classes/Interfaces/Fields
- Everything is an object
- Traits
- Singletons

Basics

```
class Account(var currentBalance:MonetaryAmount,
              val accountOwner:Person,
              val withdrawalLimit:MonetaryAmount) {

   private val txLog = new TransactionLog()
   txLog.log(“New acct with “ + currentBalance)

   def getSummary = "Name = " + accountOwner.name +
                    ", Balance = " + currentBalance

      def withdraw(amount:MonetaryAmount) {
         if (amount < currentBalance && amount < withdrawalLimit)
            currentBalance = currentBalance - amount
         else
            txLog.log("Withdrawal rejected, amount too high: " + amount)
      }
   }
```

Basics

class name/extensions
field declarations
constructor + copying to fields

```
class Account(var currentBalance:MonetaryAmount,
              val accountOwner:Person,
              val withdrawalLimit:MonetaryAmount)
              extends IAccount {

   private val txLog = new TransactionLog()
   txLog.log(“New acct with “ + currentBalance)

   def getSummary = "Name = " + accountOwner.name +
                    ", Balance = " + currentBalance

   def withdraw(amount:MonetaryAmount) {
      if (amount < currentBalance && amount < withdrawalLimit)
         currentBalance = currentBalance - amount
      else
         txLog.log("Withdrawal rejected, amount too high: " + amount)
   }
}
```

Basics

```
class Account(var currentBalance:MonetaryAmount,
              val accountOwner:Person,
              val withdrawalLimit:MonetaryAmount)
              extends IAccount {
```

- var means “variable” - readable and writeable. Automatically generates getters and setters
- val means “value” - read-only. Generates only getters and the field is not re-assignable

Basics

extra class initialization

private field

```
class Account(var currentBalance:MonetaryAmount,
              val accountOwner:Person,
              val withdrawalLimit:MonetaryAmount)
              extends IAccount {

   private val txLog = new TransactionLog()
   txLog.log(“New acct with “ + currentBalance)

   def getSummary = "Name = " + accountOwner.name +
                    ", Balance = " + currentBalance

   def withdraw(amount:MonetaryAmount) {
      if (amount < currentBalance && amount < withdrawalLimit)
         currentBalance = currentBalance - amount
      else
         txLog.log("Withdrawal rejected, amount too high: " + amount)
   }
}
```

Basics

```
class Account(var currentBalance:MonetaryAmount,
              val accountOwner:Person,
              val withdrawalLimit:MonetaryAmount) {

   private val txLog = new TransactionLog()
   txLog.log(“New acct with “ + currentBalance)

   def getSummary = "Name = " + accountOwner.name +
                    ", Balance = " + currentBalance

   def withdraw(amount:MonetaryAmount) {
      if (amount < currentBalance && amount < withdrawalLimit)
         currentBalance = currentBalance - amount
      else
         txLog.log("Withdrawal rejected, amount too high: " + amount)
   }
}
```

Basics: Accessing your fields

```
val account = new Account(balance, user, limit)
account.currentBalance = new MonetaryAmount(50)
println("New balance is " + account.currentBalance)

account.accountOwner = new Person("Hacker Johnson")
```

"Setter"

"Getter"

Won't compile - accountOwner is a "val"

```
public class Account implements IAccount {
      private MonetaryAmount currentBalance;
      private final Person accountOwner;
      private final MonetaryAmount withdrawalLimit;
      private final TransactionLog txLog;

      public Account(MonetaryAmount currentBalance,
                     Person accountOwner,
                     MonetaryAmount withdrawalLimit
                                                ) {
        this.currentBalance = currentBalance;
        this.accountOwner = accountOwner;
        this.withdrawalLimit = withdrawalLimit;
        this.txLog = new TransactionLog();
        txLog.log("New acct created");
      }

      public String getSummary() {
            return "Name = " + accountOwner.name +
                               ", Balance = " + currentBalance;
      }

      public void withdraw(MonetaryAmount amount) {
            if (amount.isLessThan(currentBalance) &&
                        amount.isLessThan(withdrawalLimit)) {
                  currentBalance = currentBalance.minus(amount);
            }
            else {
                  txLog.log("Withdrawal rejected, amount too high: "
                                                      + amount);
            }
      }

      public MonetaryAmount getCurrentBalance() {
            return currentBalance;
      }

      public void setCurrentBalance(MonetaryAmount currentBalance) {
            this.currentBalance = currentBalance;
      }

      public Person getAccountOwner() {
            return accountOwner;
      }

      public MonetaryAmount getWithdrawalLimit() {
            return withdrawalLimit;
      }
}
```

Everything is an object

- No operators
- No primitives
- No arrays
- No statics

OO:
Inheritance

```
trait Fruit {
   def getColor : String
   def hasStem : Boolean
}

class Apple extends Fruit {
   def getColor = "Red"
   def hasStem = true
}

class GrannySmithApple extends Apple {
   override def getColor = "Green"
}
```

Fruit is an ifc

Brief syntax
  single expression methods can skip {

override keyword

default is public

OO: traits

- "Interfaces with implementation"
- Like interfaces, a class can extend many traits
- Like abstract classes, a trait can define abstract and concrete methods - multiple inheritance!!

OO: traits
Special considerations

- No initialization of fields with constructors
- Special meaning of "super":
  - In Java, known when you compile the class
  - With traits, known only when you mix it in

OO: traits

- 3 interesting uses of trait
  - Rich Interfaces
  - Stackable modifications/"mixins"
  - Compose diverse functionality into a single unit

Traits: Rich Interfaces

```
trait Comparable[A] {
  def compare(that: A): Int

  def <  (that: A): Boolean = (this compare that) <  0
  def >  (that: A): Boolean = (this compare that) >  0
  def <= (that: A): Boolean = (this compare that) <= 0
  def >= (that: A): Boolean = (this compare that) >= 0
}
```


Abstract

did you notice?
 method names w/ special chars
 missing dots and parens?

this is just an abstract class, but you dont have to feel guilty about mixing it in

Traits: mixins

```
/**
 * Throw an  exception if someone calls get with
 * an undefined key
 */
trait NoNullAccessMap[K, V] extends java.util.Map[K, V] {
  abstract override def get(key: Object): V = {
    val value = super.get(key)
    assert(value != null, "No value found for key " + key)
    return value
  }
}
```

Traits: mixins

```
trait NoNullAccessMap[K, V] extends java.util.Map[K, V] {
  abstract override def get(key: Object): V = {
    val value = super.get(key)
    assert(value != null, "No value found for key " + key)
    return value
  }
}
```

The exact functionality you are overriding is not yet known

Traits: mixins

```
/**
 * Throw an exception if someone tries to call put()
 * with a key that is already in the map
 */
trait NoOverwriteMap[K, V] extends java.util.Map[K, V] {
  abstract override def put(key: K, value: V): V = {
    assert(!containsKey(key),
           "Could not set " + key + " to " + value +
           ": it is already set to " + get(key))
    return super.put(key, value)
  }
}
```

Traits: mixins (static)

```
class StrictMap[K, V] extends HashMap[K, V]
  with NoNullAccessMap[K, V]
  with NoOverwriteMap[K, V]
```

now that we're creating a real class, we use HashMap not Map

Traits: mixins (dynamic)

```
val stateForCity = new HashMap[String, String]()
                       with NoNullAccessMap[String, String]
                       with NoOverwriteMap[String, String]
```

Traits: compose functionality

```
class GetAccountsController
      extends Controllor
      with PermissionCheck
      with Logging
```

Food for thought

- If you had traits, when would you choose inheritance and when would you choose composition?

OO: singletons

```
object CheckSums {
    def calculateMD5(s:String) = "todo!"
}

def main(args:Array[String]) {
    CheckSums.calculateMD5("password")
}
```

Item 1:Consider providing static factory methods instead of contructors

concise way to define a class and "singletonize" it

update this to use an interface

OO: singletons

- Concise way to define a class AND create a singleton instance of it
- Used for "static" methods
- Thread-safe instantiation
- Extend traits or classes!
- Used exactly like any object

Scala is OO

- Scala is fairly consistent with Java in terms of simple OO concepts like classes, fields, and polymorphism
- Scala takes OO further with concepts like traits, everything-is-an-object, uniform access principle, etc.

Scala is functional

- Functions are first-class citizens
- Immutability/No Side Effects
- Pattern matching
- Type inference

Functional programming

- Imperative programs consist of *statements* that *change state*
- Functional programs evaluate *expressions* and avoid using state and mutable data

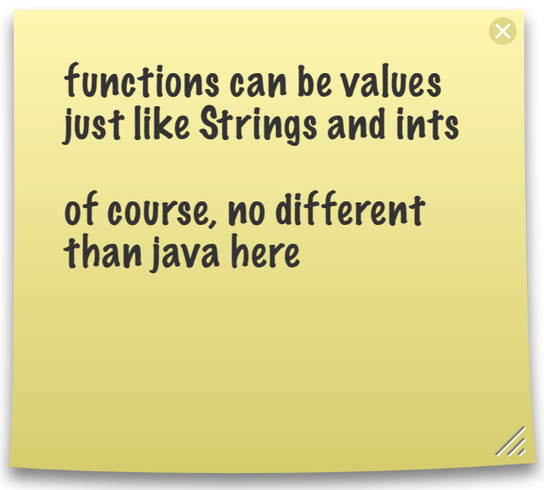
I no longer make software. I make lenses for the projection of data. :)
@nuttycom

Functional Scala: functions are objects

```
def square(x:Int) = x * x
```

Functional Scala: functions are objects

```
val squareFunction = new Function1[Int, Int]() {
  def apply(x: Int) = x * x
}
```



Functional Scala: functions are objects

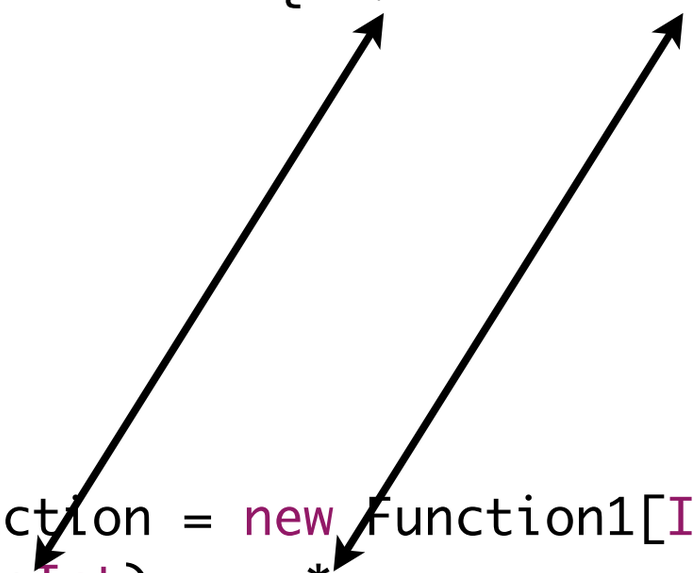
```
val squareFunction = { x:Int => x * x }
```

Is just shorthand for

```
val squareFunction = new Function1[Int, Int]() {
  def apply(x:Int) = x * x
}
```

Functional Scala: functions are objects

```
val squareFunction = { x:Int => x * x }
```



```
val squareFunction = new Function1[Int, Int]() {
  def apply(x:Int) = x * x
}
```

Imperative Java

```
public List<String> getShortCityNames(List<String cities) {
    List<String> shortCityNames = new ArrayList<String>();
    for (String city : cities) {
       if (city.length() < 8) {
          shortCityNames.add(city);
       }
    }
    return shortCityNames;
}
```

"imperative" as the opposite of "functional"

you might observe whats going on here is we're taking a list of items, and filtering out those that do not match a condition

Functional Scala

```
val cities = List("Richmond", "Charlottesville", "Ashland")
val shortCities = cities.filter { city: String => city.length < 8 }
//now shortCities has just 'Ashland", cities is unchanged
```

Passing a function into a function!

pass the transformation into the collection

Great type inference for Function types

there's "=>" again

Functional Scala

```
val shortCities = cities.filter { city => city.length < 8 }
```

more type inference - scala already knows the type of "city", since cities is a List[String]

Functional Scala

```
val shortCities = cities.filter { _.length < 8 }
```

scala uses "_" as the placeholder variable, for when naming the variable doesnt help

like "it" in groovy

all of these examples compile down to the same code

its a matter of style when considering how much inference to use

Functional Scala

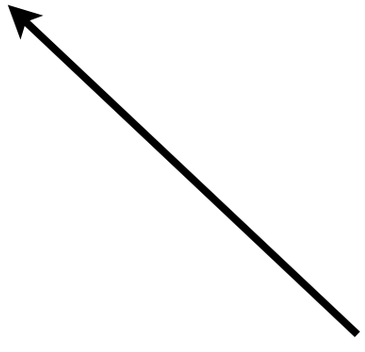
Functional style

```
val shortCities = cities.filter { _.length < 8 }
```

Imperative style

```
List<String> shortCities = getShortCityNames(cities);

public List<String> getShortCityNames(List<String cities) {
    List<String> shortCityNames = new ArrayList<String>();
    for (String city : cities) {
       if (city.length() < 8) {
          shortCityNames.add(city);
       }
    }
    return shortCityNames;
}
```

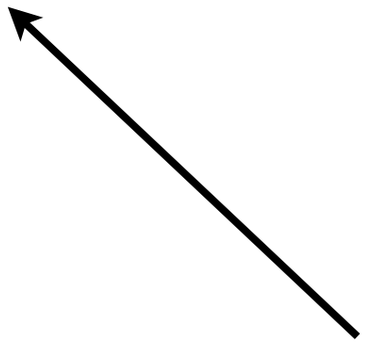


This is where bugs live

Functional Scala

Functional style

```
val shortCities = cities.filter { _.length < 8 }
```



This is where bugs live

Put the fun back in functions

```
val (minors, adults) = people partition (_.age < 18)
```

make two lists out of 1,
based on a predicate

multiple assignment

Put the fun back in functions

```
//total salary for all employees in accounting
val totalSalaryForAccountingDept = employees
                                   .filter(_.dept == "Accounting")
                                   .map(_.salary)
                                   .sum
```

Here, "map" takes a list of Employee objects, and creates a List of salaries

important to note these are creating new collections, not modifying "employees"

Put the fun back in functions

```
var clicks = 0
reactions += {
  case ButtonClicked(b) =>
    clicks += 1
    label.text = "Number of button clicks: "+ clicks
}
```

Custom control structures

```
public void transfer(Account fromAccount,
      Account toAccount,
      MonetaryAmount amount) {
   Transaction t = null;
   try {
      t = acquireTransaction();
      fromAccount.debit(amount);
      toAccount.credit(amount);
      t.commit();
   }
      catch (Exception e) {
         throw new RuntimeException("Failed to transfer ", e);
      }
      finally {
         t.close();
      }
}
```

Custom control structures

```
public void transfer(Account fromAccount,
      Account toAccount,
      MonetaryAmount amount) {
   Transaction t = null;
   try {
      t = acquireTransaction();
      fromAccount.debit(amount);
      toAccount.credit(amount);
      t.commit();
   }
      catch (Exception e) {
         t.rollback();
         throw new RuntimeException("Failed to transfer ", e);
      }
      finally {
         if (t != null) {
           t.close();
         }
      }
}
```

19 lines - where'd my code go?

```
def transfer(fromAccount:Account, toAccount:Account, amount:MonetaryAmount) {

   transaction {
      fromAccount.debit(amount)
      toAccount.credit(amount)
   }

}
```

Custom control structures

```
def transaction(transactionCode: => Unit) {
  val transaction = null
  try {
    transaction = acquireTransaction()
    transactionCode
    transaction.commit
  } catch {
    case e: Exception =>
      transaction.rollback
      throw new RuntimeException("Tx failed: rolled back", e)
  }
  finally {
      if (transaction != null) {
         transaction.close
      }
  }
}
```

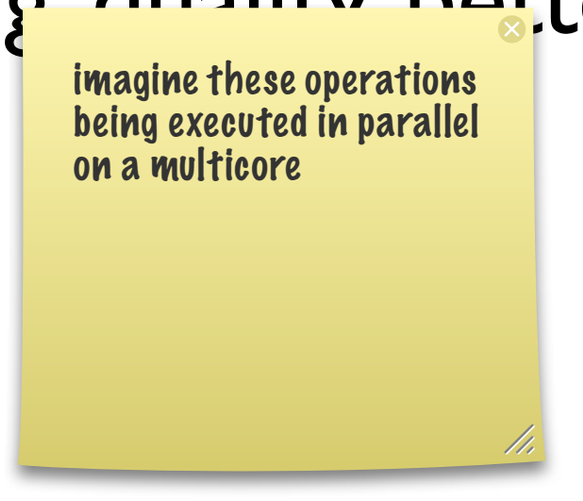
unit is scala-speak for void

this is a function that takes nothing and returns nothing

Just pass in the code that should be in TX

Why first-class functions

- Move boilerplate to library, increase readability
- Less "book-keeping" and mutable variables means less bugs
- All the advantages of libraries over boilerplate: better tuning, quality, better abstractions



I resisted list comprehensions when I first learned Python, and I resisted `filter` and `map` even longer. I insisted on making my life more difficult, sticking to the familiar way of `for` loops and `if` statements and step-by-step code-centric programming. And my Python programs looked a lot like Visual Basic programs, detailing every step of every operation in every function. And they had all the same types of little problems and obscure bugs. And it was all pointless.

— Mark Pilgrim, Dive into Python, §16.5

Scala is functional

- Functions are first-class citizens
- Immutability/No Side Effects
- Pattern matching
- Type inference

What's wrong with side effects and mutability?

- What do you hate about bad code?
  - No seperation of concerns
  - Variables changing from many different paths
  - Hard to test or reason about
  - Control flow is all over the place

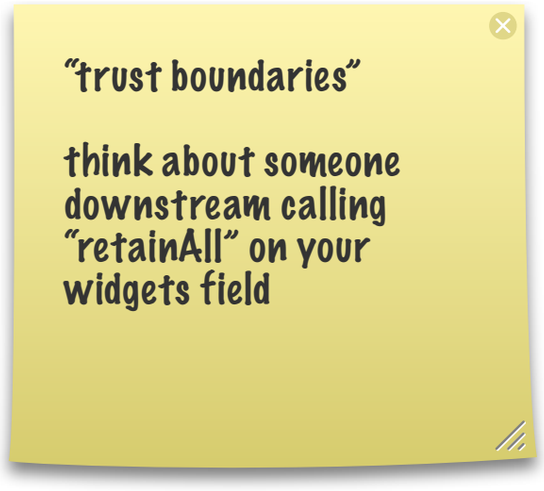
Mutability is surprising

```
Map<Widget, String> statusForWidget = new HashMap<Widget, String>();
Widget w = new Widget("Whiz", 12);
statusForWidget.put(w, "operational");
w.setName("Whizzer");
System.out.println(statusForWidget.get(w)); //Null!
```

Mutability has unintended consequences

```
private List<Widget> widgets = loadWidgets();
//......
public List<Widget> getAllWidgets() {
    return widgets;
}
```

```
thing.getAllWidgets().remove(0);
```



Mutability is hard to maintain

```
public class Account {
    private int amount;

    public Account() {
    }

    public String getSummary() {
        //can i safely use amount here?
        return "Account currently has " + amount + " dollars";
    }

    public void setAmount(int amount) {
        this.amount = amount;
    }
}
```

Side effects can blow your mind

```
public class DateParseServlet extends HttpServlet {
   private DateFormat dateParser = new SimpleDateFormat("yyyyMMdd");

   @Override
   protected void doGet(HttpServletRequest req,
                        HttpServletResponse resp) {
      resp.getWriter().print(
                  dateParser.format(req.getParameter("dateString")));
   }
}
```

Where's the bug?

item #13 in "effective java"

Why immutable and side-effect free?

- Easier to reason about
- No need to worry about what callers do with data they get from you
- We live in a multi-threaded world

Scala and Immutability

- val vs. var
- Collections library
- Actors

val vs. var

```
class Person(val first:String, val last:String)
```

- This class is immutable - first and last are read-only
- val does NOT means “immutable”: it means “non-reassignable” or “final”. Your class is only immutable if it’s “vals all the way down”

Collections library

- Scala provides two collection libraries: immutable and mutable
- immutable is “default”

Actors

- Scala preferred concurrent programming model
- No Runnable, synchronized, no locks, no shared mutable state - a “developer-centric” approach
- Modeled from Erlang

Actors

```
object Accumulator extends Actor {
  def act = {
    var sum = 0
    loop {
      react {
        case Accumulate(n) => sum += n
        case Reset => sum = 0
        case Total => reply(sum); exit
      }
    }
  }
}
```

Shared state

```
Accumulator.start
for(i <- (1 to 100)) {
    async {
       Accumulator ! Accumulate(i)
    }
}

Accumulator !? Total; match {
    case result: Int => println(result)
}
```

Lots of threads hitting the actor

actors are "shared nothing" -

messages are immutable

mutable state is private

Food for thought

- What effects does immutability have on performance?
- What effects does immutability have on correctness?

Scala is functional

- Functions are first-class citizens
- Immutability/Side Effects
- Pattern matching
- Type inference

Pattern Matching

- Pattern matching is a switch statement
  - Matching on properties of object *graphs* (not just ints)
  - Matching on object type
  - Extracting values
  - Compiler warning/RuntimeException on non-match

Pattern Matching

```
def parseBoolean(a: Any):Boolean = {
  val stringTruthies = Set("true", "1", "yes")
  a match {
    case b:Boolean => b
    case i:Int => i == 1
    case s:String => stringTruthies.contains(s)
  }
}
```

instanceof/cast
extraction of data
runtime exception if no match

remember: last expression is the return statement

Pattern Matching: Case Classes

```
case class Town(city:String, state:String)
case class Person(first:String, last:String, hometown:Town)

def getTownsThatHaveSmithsInVirginia(people:List[Person]) = {
  people.collect { person =>
      person match {
          case Person(_, "Smith", Town(town, "Virginia")) => town
          case _ =>
      }
  }
}
```

note the _ to say "any first name"

note the embedded Town object

object graph matching/ type matching/data extraction

Case classes are great for representing your "Data objects":

- Automatically define an "extractor" for pattern matching
- Automatically define equals and hashCode based on fields
- Automatically define toString

case classes are great: copy

```
val youngy = new Person("Youngy", "McBaby", 2, today)
val youngyTwin = youngy.copy(first = "Youngina")
```

Express powerful concepts concisely

```
case class Town(city:String, state:String)
case class Person(first:String, last:String, hometown:Town)
```

Converted to Java...

```
class Person {
	public static Person Person(String first, String last, Town homeTown) {
		return new Person(first, last, homeTown);
	}

	private final String first;
	private final String last;
	private final Town homeTown;

	public Person(String first, String last, Town homeTown) {
		this.first = first;
		this.last = last;
		this.homeTown = homeTown;
	}

	public String getFirst() {
		return first;
	}

	public String getLast() {
		return last;
	}

	public Town getHomeTown() {
		return homeTown;
	}

	@Override
	public int hashCode() {
		final int prime = 31;
		int result = 1;
		result = prime * result + ((first == null) ? 0 : first.hashCode());
		result = prime * result + ((homeTown == null) ? 0 : homeTown.hashCode());
		result = prime * result + ((last == null) ? 0 : last.hashCode());
		return result;
	}

	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (obj == null)
			return false;
		if (getClass() != obj.getClass())
			return false;
		Person other = (Person) obj;
		if (first == null) {
			if (other.first != null)
				return false;
		}
		else if (!first.equals(other.first))
			return false;
		if (homeTown == null) {
			if (other.homeTown != null)
				return false;
		}
		else if (!homeTown.equals(other.homeTown))
			return false;
		if (last == null) {
			if (other.last != null)
				return false;
		}
		else if (!last.equals(other.last))
			return false;
		return true;
	}

	@Override
	public String toString() {
		return "Person [first=" + first + ", homeTown=" + homeTown + ", last=" + last + "]";
	}
}
```

```
public class Town {
	public static Town Town(String city, String state) {
		return new Town(city, state);
	}
	private final String city;
	private final String state;

	public Town(String city, String state) {
		this.city = city;
		this.state = state;
	}

	public String getCity() {
		return city;
	}

	public String getState() {
		return state;
	}

	@Override
	public int hashCode() {
		final int prime = 31;
		int result = 1;
		result = prime * result + ((city == null) ? 0 : city.hashCode());
		result = prime * result + ((state == null) ? 0 : state.hashCode());
		return result;
	}

	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (obj == null)
			return false;
		if (getClass() != obj.getClass())
			return false;
		Town other = (Town) obj;
		if (city == null) {
			if (other.city != null)
				return false;
		}
		else if (!city.equals(other.city))
			return false;
		if (state == null) {
			if (other.state != null)
				return false;
		}
		else if (!state.equals(other.state))
			return false;
		return true;
	}

	@Override
	public String toString() {
		return "Town [city=" + city + ", state=" + state + "]";
	}
}
```

```java
public class Town {
	public static Town Town(String city, String state) {
		return new Town(city, state);
	}
	private final String city;
	private final String state;

	public Town(String city, String state) {
		this.city = city;
		this.state = state;
	}

	public String getCity() {
		return city;
	}

	public String getState() {
		return state;
	}

	@Override
	public int hashCode() {
		final int prime = 31;
		int result = 1;
		result = prime * result + ((city == null) ? 0 : city.hashCode());
		result = prime * result + ((state == null) ? 0 : state.hashCode());
		return result;
	}

	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (obj == null)
			return false;
		if (getClass() != obj.getClass())
			return false;
		Town other = (Town) obj;
		if (city == null) {
			if (other.city != null)
				return false;
		}
		else if (!city.equals(other.city))
			return false;
		if (state == null) {
			if (other.state != null)
				return false;
		}
		else if (!state.equals(other.state))
			return false;
		return true;
	}

	@Override
	public String toString() {
		return "Town [city=" + city + ", state=" + state + "]";
	}
}
```

Scala is functional

- Functions are first-class citizens
- Immutability/Side Effects
- Pattern matching
- Type inference

Type inference

- Local variables
- Return type
- Type parameters

More noise reduction

- Semicolons
- Dots
- Braces
- Parenthesis
- return

- How does type inference affect readability?

reducing boilerplate is obvious win
but what about inherited/old code?
scala style guide recommends never inferring return type

Extra innings

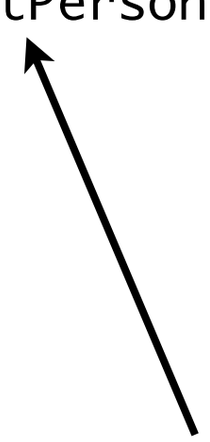
- Option
- Implicit conversions
- Named parameters/default values

Where's the bug?

```
String name = getPerson("person1234").getName();
```

Where's the bug?

```
String name = getPerson("person1234").getName();
```



Null!

How do you know when a value is null?

- Great guesser
- Clairvoyance
- Naming conventions
- Just pretend nothing will be null
- Just pretend everything might be null
- Annotations
- Use a language that supports nullable tyypes
- Documentation
- Sentinel values
- Great tester

How do you know when a value is null?

Scala's approach: Option[T]

```
def getPerson(id:String):Option[Person] = {
    if (id == "person123") {
        Some(new Person("Bob"))
    }
    else {
        None
    }
}
```

Scala's approach: Option[T]

- Option is extended by Some and None
- Encodes optional values into your API with zero magic - can be used in Java
- Scala's type inference keeps the noise to a minimum
- May seem like an extra burden, but the burden is already there once you have optional values

Scala's approach: Option[T]

```
//previous approach: doesnt compile!
val name = getPerson("person123").name

//.get will throw a RuntimeException if it is null here
val name2 = getPerson("person123").get.name

//get the value, or use a default
val name3 = getPerson("person123").getOrElse("Name unknown")

//in some cases (not this one), pattern matching is nice
val name4 = getPerson("person123") match {
    case Some(name) => name
    case None => "Name Unknown"
}
```

Implicit conversions

- Automatically convert types
- Commonly used to add behavior to existing closed source classes
- Programmer controls the scope of the conversions - not global
- Goes beyond just adding methods

Implicit Conversions
A Rich Wrapper

```
class RichInt(i:Int) {
   def times(f: => Unit) = {
      for (x <- 0 to i) f
   }
}
```

```
new RichInt(5).times { println("hey!") }
```

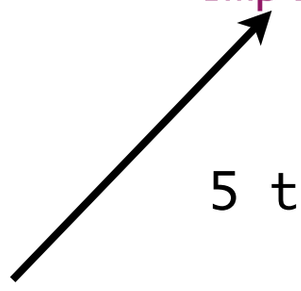
yuck

Implicit Conversions

```
class RichInt(i:Int) {
   def times(f: => Unit) = {
      for (x <- 0 to i) f
   }
}

implicit def intToRichInt(i:Int) = new RichInt(i)

5 times println("hey!")
```



Named and default parameters

```
def sendEmail(
    to : List[String],
    from : String,
    subject : String = "No subject",
    cc : List[String] = Nil,
    bcc : List[String] = Nil,
    attachments : List[File] = Nil
)
```

Named and default parameters

```
sendEmail(List("a@a.com"), "b@b.com")

sendEmail(to = List("a@a.com"),
          from = "b@b.com",
          subject ="Status Check!")

sendEmail(to = List("a@a.com"),
          from = "b@b.com",
          subject ="Status Check!",
          attachments = List(new File("coolpic.jpg")))
```

Out of time

- DSLs
- Sane generics (no wildcards!)
- Non-alpha method names
- Multi-line strings
- Structural typing
- XML Literals
- Currying

My top10 Aspects of Scala

- Java Interop
- Concise/Type inference
- Functions-as-objects
- Some/None/Option
- implicit conversions
- traits
- rich collections library
- default/named params
- case classes
- REPL

These concepts are in other statically typed JVM languages too!

- Fantom (from Richmond!)
- Gosu
- Java 8 (currently "late 2012")

Go try it!

- http://scala-lang.org
- http://www.simplyscala.com/ - try simple samples without installing
- I will be posting slides + a pre-built Eclipse with this talks code on my blog.

Thanks!

adamrabung@gmail.com
http://squirrelsewer.blogspot.com
twitter:squirrelsewer