A Brief Intro to Scala

Dynamic vs. Static

Dynamic (Ruby)

- Concise
- Scriptable
- **R**ead-**E**val-**P**rint **L**oop (irb)
- Higher Order Functions
- Extend existing classes
- Duck Typing
- method_missing

Static (Java)

- Better IDE Support
- Fewer Tests
- Documentation
- Open Source Libs
- Performance
- JVM Tools (VisualVM)
- True Multi-threading

Scala

- ✓Concise
- ✓Scriptable
- ✓**R**ead-**E**val-**P**rint **L**oop
- ✓Higher Order Functions
- ✓Extend existing classes
- ✓Duck Typing
- ✓method_missing
- ✓Better IDE Support
- ✓Fewer Tests
- ✓Documentation
- ✓Open Source Libs
- ✓Performance
- ✓JVM Tools (VisualVM)
- ✓True Multi-threading

Scalable **la**nguage

Scala is a **modern multi-paradigm** programming language designed to express **common** programming **patterns** in a **concise**, **elegant**, and **type-safe** way.

Scala

- Statically Typed
- Runs on JVM, full inter-op with Java
- Object Oriented
- Functional
- Dynamic Features

Scala is Practical

- Can be used as drop-in replacement for Java
  - Mixed Scala/Java projects
- Use existing Java libraries
- Use existing Java tools (Ant, Maven, JUnit, etc...)
- Decent IDE Support (NetBeans, IntelliJ, Eclipse)

Scala is Concise

Type Inference

```
val sum = 1 + 2 + 3

val nums = List(1, 2, 3)

val map = Map("abc" -> List(1,2,3))
```

Explicit Types

```
val sum: Int = 1 + 2 + 3

val nums: List[Int] = List(1, 2, 3)

val map: Map[String, List[Int]] = ...
```

Higher Level

```
// Java – Check if string has uppercase character
boolean hasUpperCase = false;
for(int i = 0; i < name.length(); i++) {
    if(Character.isUpperCase(name.charAt(i))) {
        hasUpperCase = true;
        break;
    }
}
```

Higher Level

```scala
// Scala
val hasUpperCase =
  name.exists(_.isUpperCase)
```

Less Boilerplate

```
// Java
public class Person {
  private String name;
  private int age;
  public Person(String name, Int age) {  // constructor
    this.name = name;
   this.age = age;
  }
  public String getName() {              // name getter
    return name;
  }
  public int getAge() {                  // age getter
    return age;
  }
  public void setName(String name) {     // name setter
    this.name = name;
  }
  public void setAge(int age) {          // age setter
    this.age = age;
  }
}
```

Less Boilerplate

```
// Scala
class Person(var name: String, var age:
  Int)
```

Less Boilerplate

```
// Scala
class Person(var name: String, private var _age:
  Int) {
  def age = _age                // Getter for age
  def age_=(newAge:Int) {  // Setter for age
    println("Changing age to: "+newAge)
    _age = newAge
  }
}
```

Variables and Values

```
// variable
var foo = "foo"
foo = "bar"  // okay

// value
val bar = "bar"
~~bar = "foo"~~  // nope
```

Scala is Object Oriented

Pure O.O.

```
// Every value is an object
1.toString


// Every operation is a method call
1 + 2 + 3  →  (1).+(2).+(3)


// Can omit . and ( )
"abc" charAt 1  →  "abc".charAt(1)
```

Classes

```
// Classes (and abstract classes) like Java
abstract class Language(val name:String) {
  override def toString = name
}

// Example implementations
class Scala extends Language("Scala")

// Anonymous class
val scala = new Language("Scala") { /* empty
  */ }
```

Traits

```
// Like interfaces in Java
trait Language {

  val name:String

  // But allow implementation
  override def toString = name
}
```

Traits

```
trait JVM {
  override def toString = super.toString+" runs on JVM" }
trait Static {
  override def toString = super.toString+" is Static" }

// Traits are stackable
class Scala extends Language with JVM with Static {
   val name = "Scala"
}

println(new Scala) → "Scala runs on JVM is Static"
```

Singleton Objects

```scala
// Replaces static methods from Java
// Can extend/implement classes & traits

object Hello {
  def world = println("Hello World"}
}

Hello.world  →  Hello World
```

Scala is Functional

First Class Functions

```
// Lightweight anonymous functions
(x:Int) => x + 1


// Calling the anonymous function
val plusOne = (x:Int) => x + 1
plusOne(5)  →  6
```

Closures

```
// plusFoo can reference any values/variables in
   scope
var foo = 1
val plusFoo = (x:Int) => x + foo

plusFoo(5)   → 6

// Changing foo changes the return value of plusFoo
foo = 5
plusFoo(5)   → 10
```

Higher Order Functions

```
val plusOne = (x:Int) => x + 1
val nums = List(1,2,3)

// map takes a function: Int => T
nums.map(plusOne)          →  List(2,3,4)

// Inline Anonymous
nums.map(x => x + 1)       →  List(2,3,4)

// Short form
nums.map(_ + 1)            →  List(2,3,4)
```

Higher Order Functions

```
val nums = List(1,2,3,4)

// A few more examples for List class
nums.exists(_ == 2)              →   true
nums.find(_ == 2)                →   Some(2)
nums.indexWhere(_ == 2)          →   1
nums.reduceLeft(_ + _)           →   10
nums.foldLeft(100)(_ + _)        →   110

// Many more in collections library
```

Higher Order Functions

```
// functions as parameters
def call(f: Int => Int) = f(1)



call(plusOne)          →  2
call(x => x + 1)       →  2
call(_ + 1)            →  2
```

Higher Order Functions

```
// functions as parameters
def each(xs: List[Int], fun: Int => Unit) {
  if(!xs.isEmpty) {
    fun(xs.head)
    each(xs.tail, fun)
  }
}

each(List(1,2,3), println)
    →  1
    →  2
    →  3
```

Higher Order Functions

```
// More complex example with generics & pattern matching

@tailrec
def each[T](xs: List[T], fun: T => Unit): Unit = xs match
   {
  case Nil =>
  case head :: tail => fun(head); each(tail, fun)
}

each(List(1,2), println)
    →  1
    →  2

each(List("foo", "bar"), println)
    →  foo
    →  bar
```

Pattern Matching

```
def what(any:Any) = any match {
  case i:Int => "It's an Int"
  case s:String => "It's a String"
  case _ => "I don't know what it is"
}
```

```
what(123)     →  "It's an Int"
what("hello") →  "It's a String"
what(false)   →  "I don't know what it
  is"
```

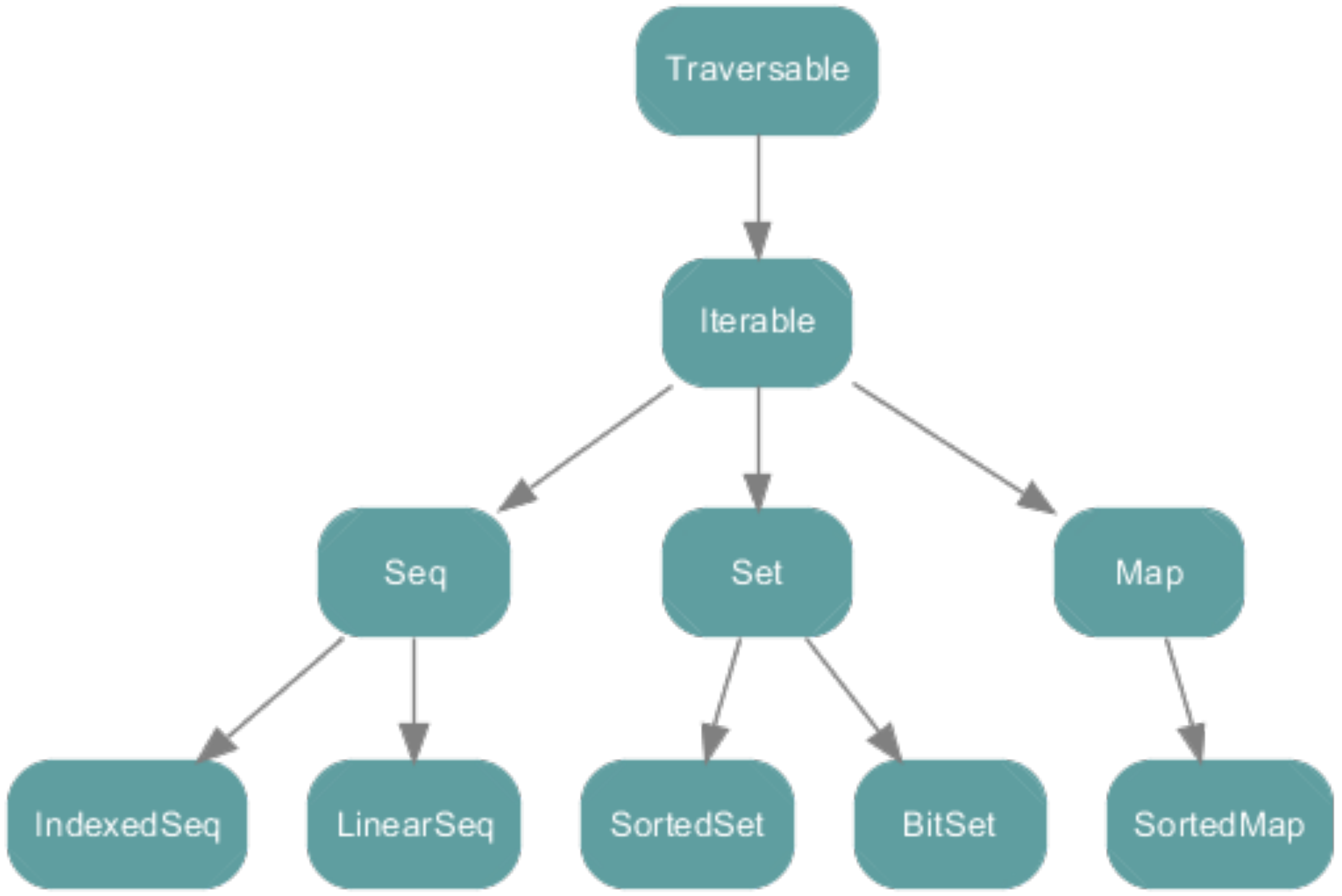
Pattern Matching

```
val nums = List(1,2,3)

// Pattern matching to create 3 vals
val List(a,b,c) = nums

a   → 1
b   → 2
c   → 3
```

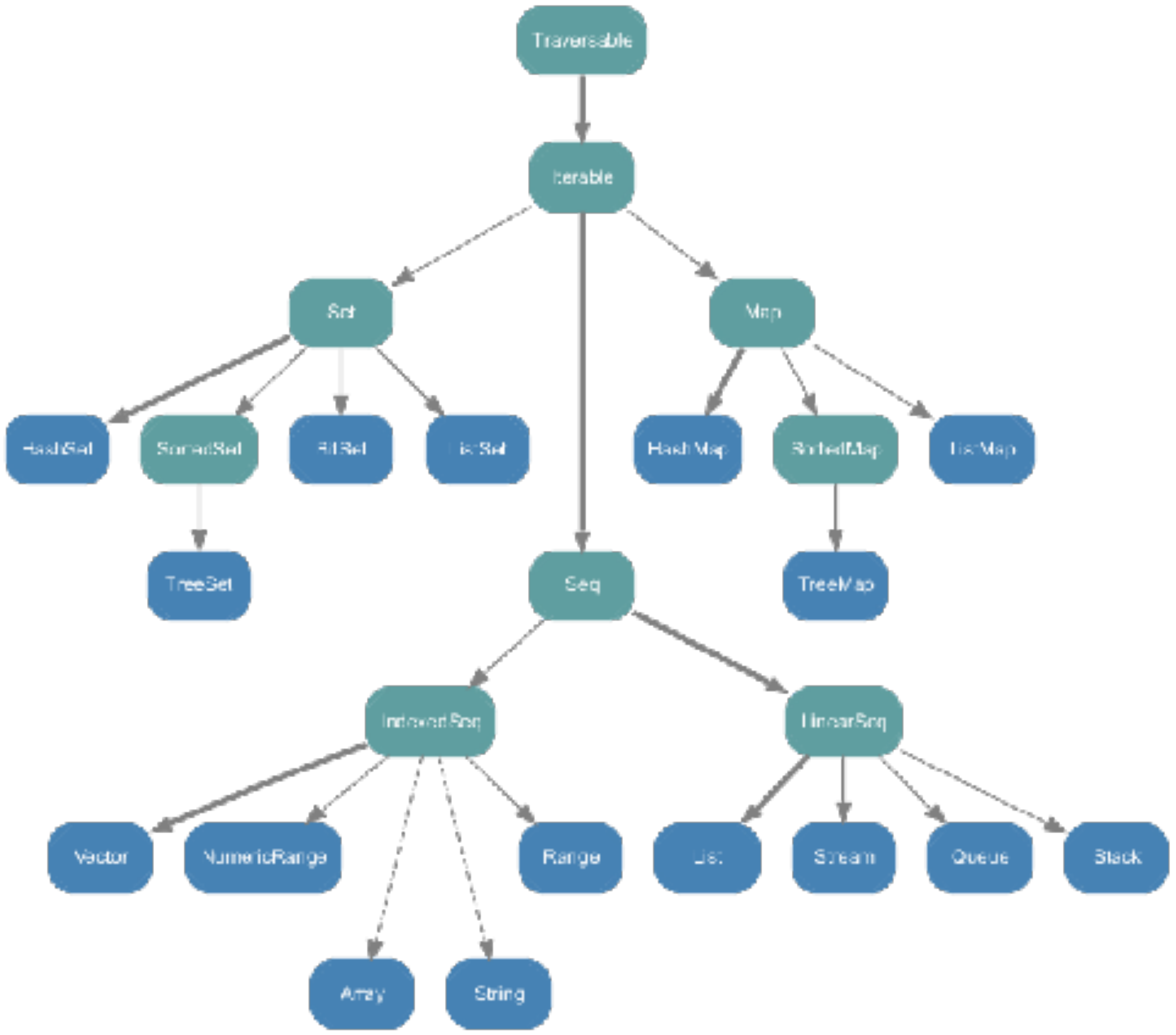
Immutable Types

```
// Immutable types by default
var nums = Set(1,2,3)
nums += 4   →   nums = nums.+(4)

// Mutable types available
import scala.collection.mutable._

val nums = Set(1,2,3)
nums += 4   →   nums.+=(4)
```

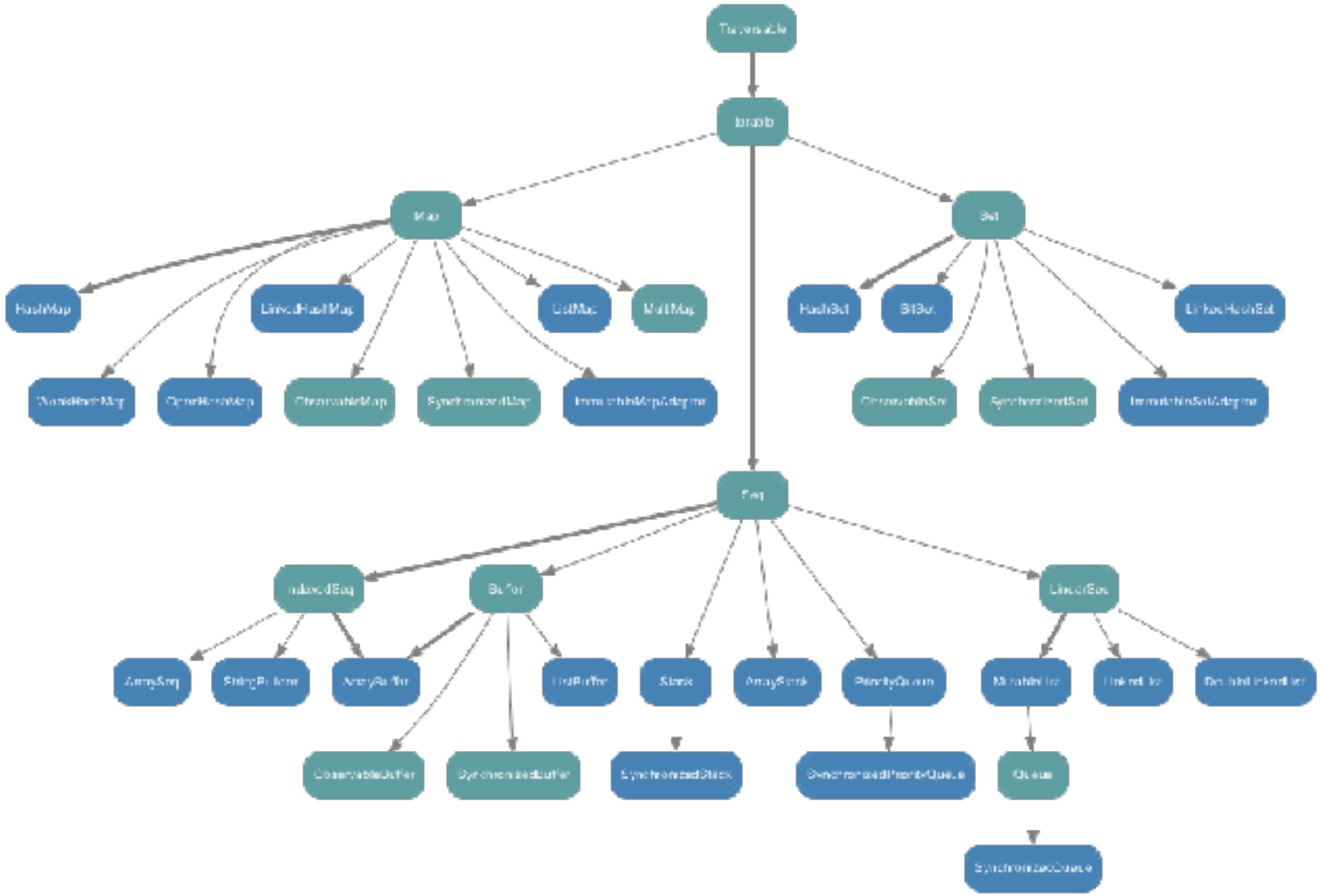
scala.collection


$$\begin{tikzcd}
 & & \text{Traversable} \arrow[d] & & \\
 & & \text{Iterable} \arrow[dl] \arrow[d] \arrow[dr] & & \\
 & \text{Seq} \arrow[dl] \arrow[d] & \text{Set} \arrow[d] \arrow[dr] & \text{Map} \arrow[dr] & \\
\text{IndexedSeq} & \text{LinearSeq} & \text{SortedSet} & \text{BitSet} & \text{SortedMap}
\end{tikzcd}$$

scala.collection.immutable



scala.collection.mutable



Or Use Existing Java Collections

- java.util
- Apache Commons Collections
- fastutil
- Trove
- Google Collections

- scala.collection.JavaConversion available to convert to and from java.util Interfaces

Scala is Dynamic

(Okay not really, but it has lots of features typically only found in Dynamic languages)

Scriptable

```
// HelloWorld.scala
println("Hello World")

bash$ scala HelloWorld.scala
Hello World

bash$ scala -e 'println("Hello World")'
Hello World
```

Read-Eval-Print Loop

```
bash$ scala
Welcome to Scala version 2.8.1.final (Java HotSpot(TM) 64-Bit
    Server VM, Java 1.6.0_22).
Type in expressions to have them evaluated.
Type :help for more information.

scala> class Foo { def bar = "baz" }
defined class Foo

scala> val f = new Foo
f: Foo = Foo@51707653

scala> f.bar
res2: java.lang.String = baz
```

Structural Typing

```
// Type safe Duck Typing
def doTalk(any:{def talk:String}) {
  println(any.talk)
}

class Duck { def talk = "Quack" }
class Dog  { def talk = "Bark"  }

doTalk(new Duck)   →   "Quack"
doTalk(new Dog)    →   "Bark"
```

Implicit Conversions

```
// Extend existing classes in a type safe way

// Goal: Add isBlank method to String class
class RichString(s:String) {
  def isBlank = null == s || "" == s.trim
}

implicit def toRichString(s:String) = new
  RichString(s)


// Our isBlank method is now available on Strings
" ".isBlank     →  true
"foo".isBlank   →  false
```

Implicit Conversions

```
// Does not type check
~~"abc".isBlank~~

// Search in-scope implicits defs that take a
// String & return a type with an isBlank method
implicit def toRichString(s:String):RichString


// Resulting code that type checks
new RichString("abc").isBlank
```

method_missing (Scala 2.9 Feature)

```
// Dynamic is a marker trait used by the compiler
class Foo extends Dynamic {
  def typed[T] = error("not implemented")

  def applyDynamic(name:String)(args:Any*) = {
    println("called: "+name+"("+args.mkString(",")
  +")")
  }

}

val f = new Foo
f.helloWorld        →   called: helloWorld()
f.hello("world")    →   called: hello(world)
f.bar(1,2,3)        →   called: bar(1,2,3)
```

Scala has tons of other cool stuff

Default Parameter Values

```
def hello(foo:Int = 0, bar:Int = 0) {
  println("foo: "+foo+"  bar: "+bar)
}
```

```
hello()        →   foo: 0  bar: 0
hello(1)       →   foo: 1  bar: 0
hello(1,2)     →   foo: 1  bar: 2
```

Named Parameters

```
def hello(foo:Int = 0, bar:Int = 0) {
  println("foo: "+foo+"  bar: "+bar)
}

hello(bar=6)         →   foo: 0   bar: 6
hello(foo=7)         →   foo: 7   bar: 0
hello(foo=8,bar=9)   →   foo: 8   bar: 9
```

Everything Returns a Value

```
val a = if(true) "yes" else "no"

val b = try{
  "foo"
} catch {
  case _ => "error"
}

val c = {
  println("hello")
  "foo"
}
```

Lazy Vals

```
// initialized on first access
lazy val foo = {
  println("init")
  "bar"
}

foo  →  init
foo  →
foo  →
```

Nested Functions

```
// Can nest multiple levels of functions
def outer() {
    var msg = "foo"
    def one() {
        def two() {
            def three() {
                println(msg)
            }
            three()
        }
        two()
    }
    one()
}
```

By-Name Parameters

```
// msg parameter automatically wrapped in closure
def log(doLog:Boolean, msg: => String) {
  if(doLog) {
    msg  // evaluates msg
    msg  // evaluates msg again!
  }
}

def foo:String = {
  println("in foo"); "Foo"
}

log(true, foo+" Bar")   // foo called twice
    →   in foo
    →   in foo

log(false, foo+" Bar")  // foo never called
```

Many More Features

- **Actors**
- **Annotations** → `@foo def hello = "world"`
- **Case Classes** → `case class Foo(bar:String)`
- **Currying** → `def foo(a:Int,b:Boolean)(c:String)`
- **For Comprehensions**
  → `for(i <- 1.to(5) if i % 2 == 0) yield i`
- **Generics** → `class Foo[T](bar:T)`
- **Package Objects**
- **Partially Applied Functions**
- **Tuples** → `val t = (1,"foo","bar")`
- **`Type Specialization`**
- **XML Literals** → `val node = <hello>world</hello>`
- **etc…**