Scala:
Growing a Language

Steven She

SE2: Software Design & Architecture

July 15, 2008

What is Scala?

Scala is...

- a general purpose programming language.

What is Scala?

Scala is...

- a general purpose programming language.
- object-oriented...

What is Scala?

Scala is...

- a general purpose programming language.
- object-oriented... and functional.

What is Scala?

Scala is...

- a general purpose programming language.
- object-oriented... and functional.
- like a dynamic language...

What is Scala?

Scala is...

- a general purpose programming language.
- object-oriented... and functional.
- like a dynamic language... but is really statically typed.

What is Scala?

Scala is...

- a general purpose programming language.
- object-oriented... and functional.
- like a dynamic language... but is really statically typed.
- seemlessly integrated with Java.

What is Scala?

Scala is...

- a general purpose programming language.
- object-oriented... and functional.
- like a dynamic language... but is really statically typed.
- (almost) seemlessly integrated with Java.

A Brief History

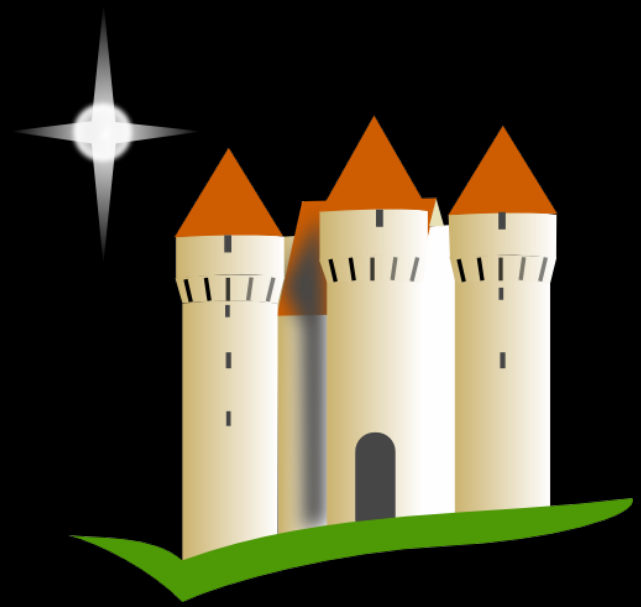
- Designed by Martin Odersky, Professor at EPFL.
- In 1998, created GJ. Implemented into Java 5 in 2004.
- GJ compiler became javac (v1.3 onward).
- Design for Scala started in 2001; first release in 2003.
- Scala version 2 was released 2006.

Growing a Language

Scala stands for "Scalable Language"

Cathedral.

- One plan, one designer.
- Long, arduous development time.
- Few changes made to the plan.
- Java, C, C++, etc.

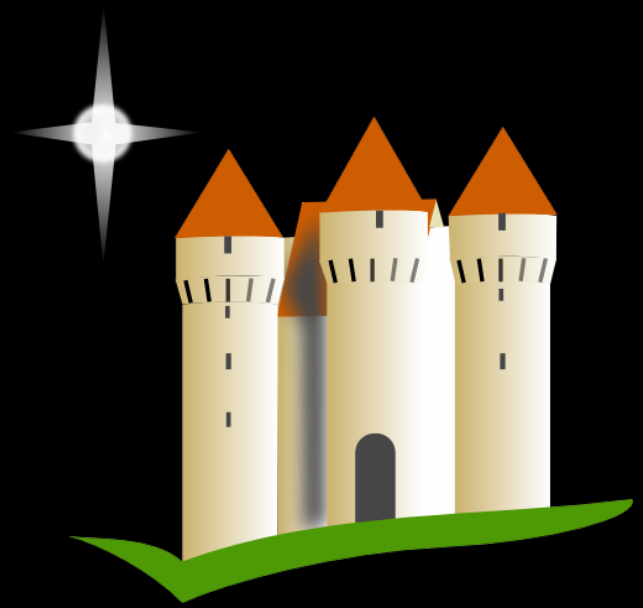


Growing a Language

Scala stands for "Scalable Language"

Cathedral.

- One plan, one designer.
- Long, arduous development time.
- Few changes made to the plan.
- Java, C, C++, etc.



Growing a Language (cont.)

Scala stands for "Scalable Language"

Bazaar.

- No single plan.
- Users contribute to the design of the system.
- Language grows as the users see fit.
- Scala was designed to support a bazaar.

[9] http://catb.org/esr/writings/cathedral-bazaar/cathedral-bazaar/

Programming in a Bazaar

Framework code should appear indistinguishable from **native language features** .

Something like...*Scheme*?

Scala is a *practical* language that enables users to grow the language in a seamless manner.

Programming in a Bazaar

Framework code should appear indistinguishable from **native language features** .

Something like...*Scheme*?

Scala is a *practical* language that enables users to grow the language in a seamless manner.

Programming in a Bazaar

Framework code should appear indistinguishable from **native language features** .

Something like...*Scheme*?

Scala is a *practical* language that enables users to grow the language in a seamless manner.

Let's talk about Scala!

Scala > Java

Topic for this talk:

What language constructs does Scala provide that enable a user to grow the language?

Let's talk about Scala!

Scala > Java

Topic for this talk:

What language constructs does Scala provide that enable a user to grow the language?

Outline

Outline

Hello, world!

```
object HelloWorld {
  def main(args : Array[String]) : Unit = {
    println("Hello, world!")
  }
}
```

Hello, world!

Singleton

```
object HelloWorld {
  def main(args : Array[String]) : Unit = {
    println("Hello, world!")
  }
}
```

Hello, world!


```
object HelloWorld {
  def main(args : Array[String]) : Unit = {
    println("Hello, world!")
  }
}
```

Hello, world!


```
object HelloWorld {
  def main(args : Array[String]) : Unit = {
    println("Hello, world!")
  }
}
```

Hello, world!

```
object HelloWorld {
  def main(args : Array[String]) : Unit = {
    println("Hello, world!")
  }
}
```

Hello, world!

```
object HelloWorld {
  def main(args : Array[String]) : Unit = {
    println("Hello, world!")
  }
}
```

Predef.println(...)

Basic Method Declarations

```
object HelloWorld {
  def simple : String =
    "Hello World!"

  def verbose() : String = {
    return "Hello World!"
  }

  def ++() : String =
    { val str = "Hello World!"; str }

  def infer = "Hello World!"
}
```

Basic Method Declarations

```
object HelloWorld {
  def simple : String =
    "Hello World!"

  def verbose() : String = {
    return "Hello World!"
  }

  def ++() : String =
    { val str = "Hello World!"; str }

  def infer = "Hello World!"
}
```

Basic Method Declarations

```
object HelloWorld {
  def simple : String =
    "Hello World!"

  def verbose() : String = {
    return "Hello World!"
  }

  def ++() : String =
    { val str = "Hello World!"; str }

  def infer = "Hello World!"
}
```

Basic Method Declarations

```
object HelloWorld {
  def simple : String =
    "Hello World!"

  def verbose() : String = {
    return "Hello World!"
  }

  def ++() : String =
    { val str = "Hello World!"; str }

  def infer = "Hello World!"
}
```

Basic Method Declarations

```
object HelloWorld {
  def simple : String =
    "Hello World!"

  def verbose() : String = {
    return "Hello World!"
  }

  def ++() : String =
    { val str = "Hello World!"; str }

  def infer = "Hello World!"
}
```

Type Less With Type Inference

```
def infer = "Hello World!" //returns String

val strList =
    "a" :: "b" :: "c" :: Nil //List[String]

val intList =
    1 :: 2 :: 3 :: Nil //List[Int]

val anyList =
    1  :: 'b' :: "c" :: Nil //List[Any]
```

Type Less With Type Inference

```
def infer = "Hello World!" //returns String

val strList =
    "a" :: "b" :: "c" :: Nil //List[String]

val intList =
    1 :: 2 :: 3 :: Nil //List[Int]

val anyList =
    1  :: 'b' :: "c" :: Nil //List[Any]
```

Type Less With Type Inference

```
def infer = "Hello World!" //returns String

val strList =
    "a" :: "b" :: "c" :: Nil //List[String]

val intList =
    1 :: 2 :: 3 :: Nil //List[Int]

val anyList =
    1  :: 'b' :: "c" :: Nil //List[Any]
```

Type Less With Type Inference

```
def infer = "Hello World!" //returns String

val strList =
    "a" :: "b" :: "c" :: Nil //List[String]

val intList =
    1 :: 2 :: 3 :: Nil //List[Int]

val anyList =
    1  :: 'b' :: "c" :: Nil //List[Any]
```

Operators are Method Calls

```
w && x || y && z ⟷ ( w.&&(x) ).||( (y).&&(z) )
```

Right-associative operators are defined with : as a suffix.

```
"a" :: "b" :: "c" :: Nil
Nil.::("c").::("b").::("a")
```

Primitives in Java are treated as objects in Scala.

int → scala.Int
boolean → scala.Boolean
...

Operators are Method Calls

```
w && x || y && z ⟷ ( w.&&(x) ).||( (y).&&(z) )
```

Right-associative operators are defined with : as a suffix.

```
"a" :: "b" :: "c" :: Nil
Nil.::("c").::("b").::("a")
```

Primitives in Java are treated as objects in Scala.

int → scala.Int
boolean → scala.Boolean
...

Operators are Method Calls

```
w && x || y && z ⟷ ( w.&&(x) ).||( (y).&&(z) )
```

Right-associative operators are defined with : as a suffix.

```
"a" :: "b" :: "c" :: Nil
Nil.::("c").::("b").::("a")
```

Primitives in Java are treated as objects in Scala.

int → scala.Int
boolean → scala.Boolean
...

Operators are Method Calls

```
w && x || y && z ⟷ ( w.&&(x) ).||( (y).&&(z) )
```

Right-associative operators are defined with : as a suffix.

```
"a" :: "b" :: "c" :: Nil
Nil.::("c").::("b").::("a")
```

Primitives in Java are treated as objects in Scala.

int → scala.Int
boolean → scala.Boolean
...

Objects, Classes and Traits

Singletons (`object`).

No statics in Scala. Singletons can be referenced and assigned.

Classes.

Same as in Java. But, they can mix in multiple **traits**.

Traits.

Similar to a Java interface, but with default implementation. A form of *mixin* where a trait is merged into a base class.

Objects, Classes and Traits

Singletons (`object`).

No statics in Scala. Singletons can be referenced and assigned.

Classes.

Same as in Java. But, they can mix in multiple **traits**.

Traits.

Similar to a Java interface, but with default implementation.
A form of *mixin* where a trait is merged into a base class.

Comparison Trait

```
trait Ordered[A] {
  def < (that: A): Boolean
  def <=(that: A) = (this < that) || (this == that)
  def > (that: A) = !(this <= that)
  def >=(that: A) = !(this < that)
}
```

```
class ComplexNumber(val real : Int, val img : Int)
    extends Number(real) with Ordered[ComplexNumber] {

  def < (that: ComplexNumber) = real < that.real ||
    real == that.real && (img < that.img)
}
```

[9] http://www.scala-lang.org/docu/files/ScalaTutorial.pdf

Comparison Trait

Type Parameter

```
trait Ordered[A] {
  def < (that: A): Boolean
  def <=(that: A) = (this < that) || (this == that)
  def > (that: A) = !(this <= that)
  def >=(that: A) = !(this < that)
}
```

```
class ComplexNumber(val real : Int, val img : Int)
    extends Number(real) with Ordered[ComplexNumber] {

  def < (that: ComplexNumber) = real < that.real ||
    real == that.real && (img < that.img)
}
```

[9] http://www.scala-lang.org/docu/files/ScalaTutorial.pdf

Comparison Trait


```
trait Ordered[A] {
  def < (that: A): Boolean
  def <=(that: A) = (this < that) || (this == that)
  def > (that: A) = !(this <= that)
  def >=(that: A) = !(this < that)
}
```

```
class ComplexNumber(val real : Int, val img : Int)
    extends Number(real) with Ordered[ComplexNumber] {

  def < (that: ComplexNumber) = real < that.real ||
    real == that.real && (img < that.img)
}
```

[9] http://www.scala-lang.org/docu/files/ScalaTutorial.pdf

Comparison Trait

Pre-defined .equals method

```
trait Ordered[A] {
  def < (that: A): Boolean
  def <=(that: A) = (this < that) || (this == that)
  def > (that: A) = !(this <= that)
  def >=(that: A) = !(this < that)
}
```

```
class ComplexNumber(val real : Int, val img : Int)
    extends Number(real) with Ordered[ComplexNumber] {

  def < (that: ComplexNumber) = real < that.real ||
    real == that.real && (img < that.img)
}
```

[o] http://www.scala-lang.org/docu/files/ScalaTutorial.pdf

Comparison Trait

```
trait Ordered[A] {
  def < (that: A): Boolean
  def <=(that: A) = (this < that) || (this == that)
  def > (that: A) = !(this <= that)
  def >=(that: A) = !(this < that)
}
```

Boolean operators

```
class ComplexNumber(val real : Int, val img : Int)
    extends Number(real) with Ordered[ComplexNumber] {

  def < (that: ComplexNumber) = real < that.real ||
    real == that.real && (img < that.img)
}
```

[o] http://www.scala-lang.org/docu/files/ScalaTutorial.pdf

Comparison Trait

```
trait Ordered[A] {
  def < (that: A): Boolean
  def <=(that: A) = (this < that) || (this == that)
  def > (that: A) = !(this <= that)
  def >=(that: A) = !(this < that)
}

class ComplexNumber(val real : Int, val img : Int)
    extends Number(real) with Ordered[ComplexNumber] {

  def < (that: ComplexNumber) = real < that.real ||
    real == that.real && (img < that.img)
}
```

[9] http://www.scala-lang.org/docu/files/ScalaTutorial.pdf

Logging Trait

```
import org.slf4j.{Logger, LoggerFactory}

trait Logging {
  private val log = LoggerFactory.getLogger(getClass)

  def debug(msg: String, e: Throwable)  = log.debug(msg, e)
  //…
}

class ComplexNumber(real : Int, img : Int) extends Number(real)
    with Ordered[ComplexNumber] with Logging {

  //…
  debug("Error occurred!", e);
}
```

[o] http://johlrogge.wordpress.com/2009/06/27/loggingtools-in-scala/

Logging Trait

```
import org.slf4j.{Logger, LoggerFactory}

trait Logging {
  private val log = LoggerFactory.getLogger(getClass)

  def debug(msg: String, e: Throwable)  = log.debug(msg, e)
  //…
}

class ComplexNumber(real : Int, img : Int) extends Number(real)
    with Ordered[ComplexNumber] with Logging {

  //…
  debug("Error occurred!", e);
}
```

[9] http://johlrogge.wordpress.com/2009/06/27/loggingtools-in-scala/

Outline

Java Maps

```
Map<String, String> capitals = new HashMap<String, String>();

capitals.put("Ontario", "Toronto");
capitals.put("Quebec",  "Quebec City");
//...
capitals.put("BC",      "Victoria");

String toronto = capitals.get("Ontario");
String unknown = capitals.get("123"); //returns null
```

- Map implementation is explicitly instantiated.
- Map must be initialized separately (mutable!).
- Retrieving an absent key returns null.

Java Maps

```
Map<String, String> capitals = new HashMap<String, String>();

capitals.put("Ontario", "Toronto");
capitals.put("Quebec",  "Quebec City");
//...
capitals.put("BC",      "Victoria");

String toronto = capitals.get("Ontario");
String unknown = capitals.get("123"); //returns null
```

- Map implementation is explicitly instantiated.
- Map must be initialized separately (mutable!).
- Retrieving an absent key returns null.

Associative Maps

```
val capitals = Map("Ontario" -> "Toronto",
                   "Quebec"  -> "Quebec City",
                   //...
                   "BC"      -> "Victoria")
```

- Immutable, associative map.
- Written using no special keywords or syntax!
- Uses *implicit methods* and *operator methods*.

Associative Maps

```
val capitals = Map("Ontario" -> "Toronto",
                   "Quebec"  -> "Quebec City",
                   //...
                   "BC"      -> "Victoria")
```

- Immutable, associative map.
- Written using no special keywords or syntax!
- Uses *implicit methods* and *operator methods*.

Map Factory

Map("Ontario" -> "Toronto")

```
object Map {
  def apply[A, B](elems: (A, B)*) = empty[A, B] ++ elems
  def empty[A, B]: Map[A, B] = new EmptyMap[A, B]
}


trait Map[A, +B] extends …  {
  def ++ [B1 >: B] (kvs: Iterable[(A, B1)]): Map[A, B1] =
    ((this: Map[A, B1]) /: kvs) ((m, kv) => m + kv)

  def + [B1 >: B] (kv: (A, B1)): Map[A, B1] =
    update(kv._1, kv._2)
}
```

Map Factory

Map("Ontario" -> "Toronto")

```
object Map {
  def apply[A, B](elems: (A, B)*) = empty[A, B] ++ elems
  def empty[A, B]: Map[A, B] = new EmptyMap[A, B]
}
```

```
trait Map[A, +B] extends …  {
  def ++ [B1 >: B] (kvs: Iterable[(A, B1)]): Map[A, B1] =
    ((this: Map[A, B1]) /: kvs) ((m, kv) => m + kv)

  def + [B1 >: B] (kv: (A, B1)): Map[A, B1] =
    update(kv._1, kv._2)
}
```

Map Factory

Map("Ontario" -> "Toronto")

```
object Map {
  def apply[A, B](elems: (A, B)*) = empty[A, B] ++ elems
  def empty[A, B]: Map[A, B] = new EmptyMap[A, B]
}
```

```
trait Map[A, +B] extends …  {
  def ++ [B1 >: B] (kvs: Iterable[(A, B1)]): Map[A, B1] =
    ((this: Map[A, B1]) /: kvs) ((m, kv) => m + kv)

  def + [B1 >: B] (kv: (A, B1)): Map[A, B1] =
    update(kv._1, kv._2)
}
```

Map Factory

Map("Ontario" -> "Toronto")

```
object Map {
  def apply[A, B](elems: (A, B)*) = empty[A, B] ++ elems
  def empty[A, B]: Map[A, B] = new EmptyMap[A, B]
}
```

```
trait Map[A, +B] extends …  {
  def ++ [B1 >: B] (kvs: Iterable[(A, B1)]): Map[A, B1] =
    ((this: Map[A, B1]) /: kvs) ((m, kv) => m + kv)

  def + [B1 >: B] (kv: (A, B1)): Map[A, B1] =
    update(kv._1, kv._2)
}
```

Map Factory

Map("Ontario" -> "Toronto")

```
object Map {
  def apply[A, B](elems: (A, B)*) = empty[A, B] ++ elems
  def empty[A, B]: Map[A, B] = new EmptyMap[A, B]
}
```

```
trait Map[A, +B] extends …  {
  def ++ [B1 >: B] (kvs: Iterable[(A, B1)]): Map[A, B1] =
    ((this: Map[A, B1]) /: kvs) ((m, kv) => m + kv)

  def + [B1 >: B] (kv: (A, B1)): Map[A, B1] =
    update(kv._1, kv._2)
}
```

Map Factory

Map("Ontario" -> "Toronto")

```
object Map {
  def apply[A, B](elems: (A, B)*) = empty[A, B] ++ elems
  def empty[A, B]: Map[A, B] = new EmptyMap[A, B]
}
```

```
trait Map[A, +B] extends …  {
  def ++ [B1 >: B] (kvs: Iterable[(A, B1)]): Map[A, B1] =
    ((this: Map[A, B1]) /: kvs) ((m, kv) => m + kv)

  def + [B1 >: B] (kv: (A, B1)): Map[A, B1] =
    update(kv._1, kv._2)
}
```

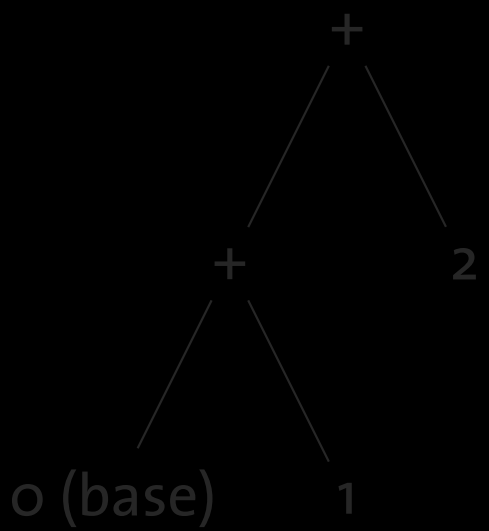
Aside: Folding

```
(0 /: (1 until 10))( (sum,num) => sum + num )
```

```
(1 until 10).foldLeft(0)( (sum,num) => sum + num )
```

```
var sum = 0
for (num <- 1 until 10)
  sum = sum + num
```

```
(0 /: (1 until 10))( _ + _ )
```



Aside: Folding

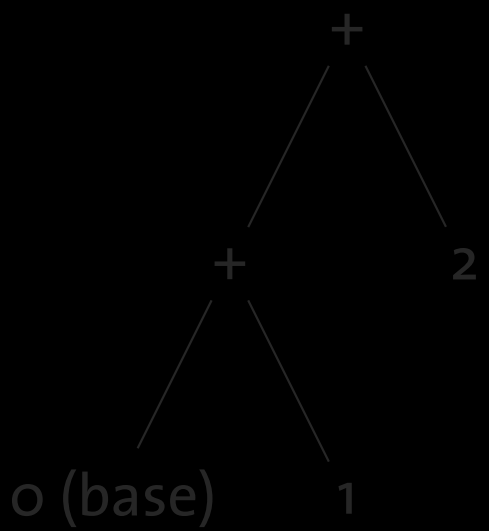
Returns a List[Int]

```
(0 /: (1 until 10))( (sum,num) => sum + num )
```

```
(1 until 10).foldLeft(0)( (sum,num) => sum + num )
```

```
var sum = 0
for (num <- 1 until 10)
  sum = sum + num
```

```
(0 /: (1 until 10))( _ + _ )
```



Aside: Folding

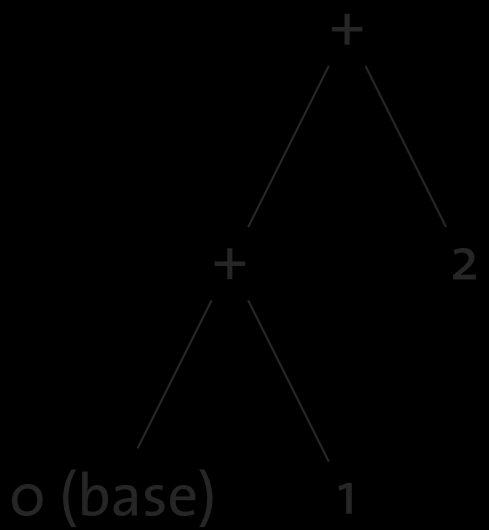
Anonymous function

```
(0 /: (1 until 10))( (sum,num) => sum + num )
```

```
(1 until 10).foldLeft(0)( (sum,num) => sum + num )
```

```
var sum = 0
for (num <- 1 until 10)
  sum = sum + num
```

```
(0 /: (1 until 10))( _ + _ )
```



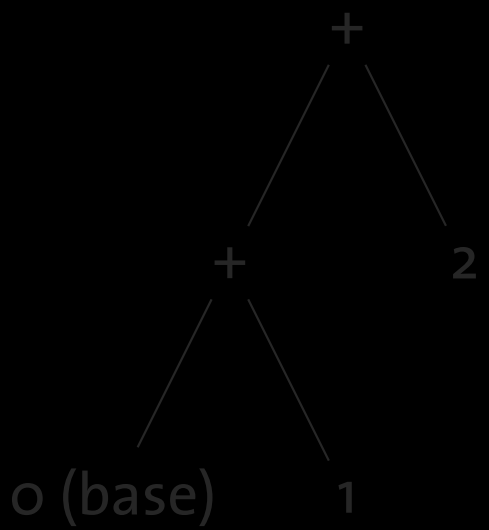
Aside: Folding

```
(0 /: (1 until 10))( (sum,num) => sum + num )

(1 until 10).foldLeft(0)( (sum,num) => sum + num )
```

```
var sum = 0
for (num <- 1 until 10)
  sum = sum + num

(0 /: (1 until 10))( _ + _ )
```



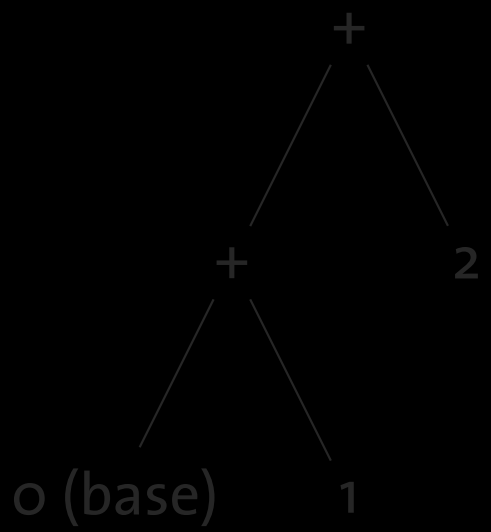
Aside: Folding

```
(0 /: (1 until 10))( (sum,num) => sum + num )

(1 until 10).foldLeft(0)( (sum,num) => sum + num )
```

```
var sum = 0
for (num <- 1 until 10)
  sum = sum + num
```

```
(0 /: (1 until 10))( _ + _ )
```



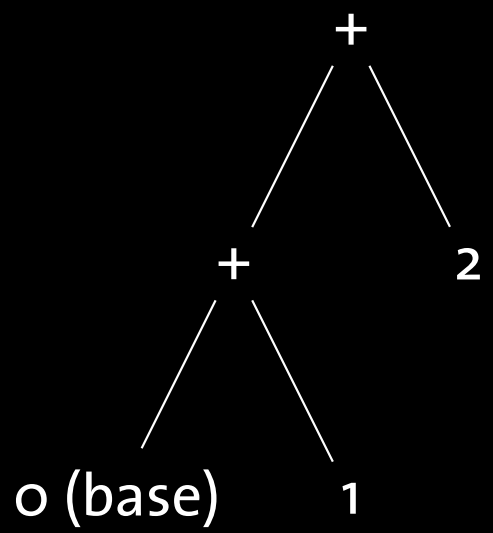
Aside: Folding

```
(0 /: (1 until 10))( (sum,num) => sum + num )

(1 until 10).foldLeft(0)( (sum,num) => sum + num )
```

```
var sum = 0
for (num <- 1 until 10)
  sum = sum + num

(0 /: (1 until 10))( _ + _ )
```



Map Factory

Map("Ontario" -> "Toronto")

```
object Map {
  def apply[A, B](elems: (A, B)*) = empty[A, B] ++ elems
  def empty[A, B]: Map[A, B] = new EmptyMap[A, B]
}
```

```
trait Map[A, +B] extends …  {
  def ++ [B1 >: B] (kvs: Iterable[(A, B1)]): Map[A, B1] =
    ((this: Map[A, B1]) /: kvs) ((m, kv) => m + kv)

  def + [B1 >: B] (kv: (A, B1)): Map[A, B1] =
    update(kv._1, kv._2)
}
```

Map Factory

Map("Ontario" -> "Toronto")

```
object Map {
  def apply[A, B](elems: (A, B)*) = empty[A, B] ++ elems
  def empty[A, B]: Map[A, B] = new EmptyMap[A, B]
}
```

```
trait Map[A, +B] extends …  {
  def ++ [B1 >: B] (kvs: Iterable[(A, B1)]): Map[A, B1] =
    ((this: Map[A, B1]) /: kvs) ((m, kv) => m + kv)

  def + [B1 >: B] (kv: (A, B1)): Map[A, B1] =
    update(kv._1, kv._2)
}
```

Map Factory

Map("Ontario" -> "Toronto")

```
object Map {
  def apply[A, B](elems: (A, B)*) = empty[A, B] ++ elems
  def empty[A, B]: Map[A, B] = new EmptyMap[A, B]
}
```

```
trait Map[A, +B] extends …  {
  def ++ [B1 >: B] (kvs: Iterable[(A, B1)]): Map[A, B1] =
    ((this: Map[A, B1]) /: kvs) ((m, kv) => m + kv)

  def + [B1 >: B] (kv: (A, B1)): Map[A, B1] =
    update(kv._1, kv._2)
}
```

Specializing the Map

```
class EmptyMap[...] extends Map[...] {
  def update [B1 >: B](key: A, value: B1): Map[A, B1] =
    new Map1(key, value)
}

class Map1 extends Map[...] {
  def update [B1 >: B](key: A, value: B1): Map[A, B1] =
    if (key == key1) new Map1(key1, value)
    else new Map2(key1, value1, key, value)
}
```

- More efficient than just a Hash Map.
- Factory pattern is built into the language.

Specializing the Map

```
class EmptyMap[...] extends Map[...] {
  def update [B1 >: B](key: A, value: B1): Map[A, B1] =
    new Map1(key, value)
}

class Map1 extends Map[...] {
  def update [B1 >: B](key: A, value: B1): Map[A, B1] =
    if (key == key1) new Map1(key1, value)
    else new Map2(key1, value1, key, value)
}
```

- More efficient than just a Hash Map.
- Factory pattern is built into the language.

Arrow on a String

```
Map( "Ontario" .->( "Toronto" ))
```

- …but, `java.lang.String` doesn't have a `->` method!
- enter *implicit methods*.

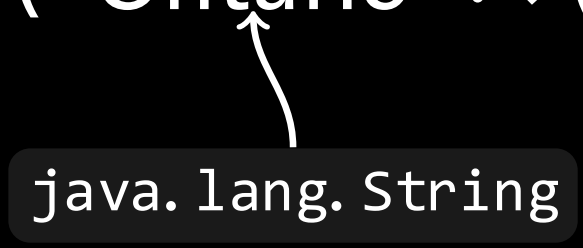
Arrow on a String

```
Map( "Ontario" .->( "Toronto" ))
```

- …but, `java.lang.String` doesn't have a `->` method!
- enter *implicit methods*.

Arrow on a String

```
Map( "Ontario" .->( "Toronto" ))
```



- …but, `java. lang. String` doesn't have a `->` method!
- enter *implicit methods.*

Arrow on a String

```
Map( "Ontario" .->( "Toronto" ))
```

- ...but, `java.lang.String` doesn't have a `->` method!
- enter *implicit methods*.

Arrow on a String

```
Map( "Ontario" .->( "Toronto" ))
```

- ...but, `java.lang.String` doesn't have a `->` method!
- enter *implicit methods.*

Arrow on a String

```
Map( "Ontario" .->( "Toronto" ))
```

- ...but, `java.lang.String` doesn't have a `->` method!
- enter *implicit methods*.

Implicit Methods

Definition

An *implicit method* is automatically called when an object of type *A* is called, but an object of type *B* is required.

```
implicit def str2int(s : String) : Int
  = Integer.parseInt(s)

def addOne(num : Int) = num + 1

val result = addOne("1")
println(result)
```

Implicit Methods

Definition

An *implicit method* is automatically called when an object of type *A* is called, but an object of type *B* is required.

```
implicit def str2int(s : String) : Int
  = Integer.parseInt(s)

def addOne(num : Int) = num + 1

val result = addOne("1")
println(result)
```

Implicit Methods

Definition

An *implicit method* is automatically called when an object of type *A* is called, but an object of type *B* is required.

```
implicit def str2int(s : String) : Int
  = Integer.parseInt(s)

def addOne(num : Int) = num + 1

val result = addOne("1")
println(result)
```

Implicit Methods

Definition

An *implicit method* is automatically called when an object of type *A* is called, but an object of type *B* is required.

```
implicit def str2int(s : String) : Int
  = Integer.parseInt(s)

def addOne(num : Int) = num + 1

val result = addOne("1")
println(result)
```

Implicit Methods

Definition

An *implicit method* is automatically called when an object of type *A* is called, but an object of type *B* is required.

```
implicit def str2int(s : String) : Int
  = Integer.parseInt(s)

def addOne(num : Int) = num + 1

val result = addOne("1")
println(result)
```

Implicit Methods (cont.)

Implicit methods could also be used to *introduce* new methods into existing classes.

```
class PigLatin(x : String) {
  def pigSpeak = x.substring(1) + "-" + x(0) + "ay"
}

implicit def str2pigLatin(s : String)
  = new PigLatin(s)

println( "hello".pigSpeak ) //outputs "ello-hay"
```

Implicit Methods (cont.)

Implicit methods could also be used to *introduce* new methods into existing classes.

```
class PigLatin(x : String) {
  def pigSpeak = x.substring(1) + "-" + x(0) + "ay"
}

implicit def str2pigLatin(s : String)
  = new PigLatin(s)

println( "hello".pigSpeak ) //outputs "ello-hay"
```

Implicit Methods (cont.)

Implicit methods could also be used to *introduce* new methods into existing classes.

```
class PigLatin(x : String) {
  def pigSpeak = x.substring(1) + "-" + x(0) + "ay"
}

implicit def str2pigLatin(s : String)
  = new PigLatin(s)

println( "hello".pigSpeak ) //outputs "ello-hay"
```

Implicit Methods (cont.)

Implicit methods could also be used to *introduce* new methods into existing classes.

```
class PigLatin(x : String) {
  def pigSpeak = x.substring(1) + "-" + x(0) + "ay"
}

implicit def str2pigLatin(s : String)
  = new PigLatin(s)

println( "hello".pigSpeak ) //outputs "ello-hay"
```

Adding an Arrow to String

Map("Ontario" -> "Toronto")

```
class ArrowAssoc[A](x: A) {
    def -> [B](y: B): Tuple2[A, B] = (x, y)
}

implicit def any2ArrowAssoc[A](x: A)
    = new ArrowAssoc(x)

    "Ontario".->("Toronto")
⟹ new ArrowAssoc("Ontario").->("Toronto")
⟹ ("Ontario", "Toronto")
```

Adding an Arrow to String

Map("Ontario" -> "Toronto")

```
class ArrowAssoc[A](x: A) {
    def -> [B](y: B): Tuple2[A, B] = (x, y)
}

implicit def any2ArrowAssoc[A](x: A)
    = new ArrowAssoc(x)

    "Ontario".->("Toronto")
⟹ new ArrowAssoc("Ontario").->("Toronto")
⟹ ("Ontario", "Toronto")
```

Adding an Arrow to String

Map("Ontario" -> "Toronto")

```
class ArrowAssoc[A](x: A) {
    def -> [B](y: B): Tuple2[A, B] = (x, y)
}

implicit def any2ArrowAssoc[A](x: A)
    = new ArrowAssoc(x)
```

"Ontario".->("Toronto")

⟹ new ArrowAssoc("Ontario").->("Toronto"))

⟹ ("Ontario", "Toronto")

Adding an Arrow to String

Map("Ontario" -> "Toronto")

```
class ArrowAssoc[A](x: A) {
    def -> [B](y: B): Tuple2[A, B] = (x, y)
}

implicit def any2ArrowAssoc[A](x: A)
    = new ArrowAssoc(x)
```

- "Ontario".->("Toronto")
- ↪ new ArrowAssoc("Ontario").->("Toronto"))
- ↪ ("Ontario", "Toronto")

Adding an Arrow to String

Map("Ontario" -> "Toronto")

```
class ArrowAssoc[A](x: A) {
    def -> [B](y: B): Tuple2[A, B] = (x, y)
}

implicit def any2ArrowAssoc[A](x: A)
    = new ArrowAssoc(x)
```

- "Ontario".->("Toronto")
- ↪ new ArrowAssoc("Ontario").->("Toronto"))
- ↪ ("Ontario", "Toronto")

Adding an Arrow to String

Map("Ontario" -> "Toronto")

```
class ArrowAssoc[A](x: A) {
    def -> [B](y: B): Tuple2[A, B] = (x, y)
}

implicit def any2ArrowAssoc[A](x: A)
    = new ArrowAssoc(x)
```

- "Ontario".->("Toronto")
- $\hookrightarrow$ new ArrowAssoc("Ontario").->("Toronto"))
- $\hookrightarrow$ ("Ontario", "Toronto")

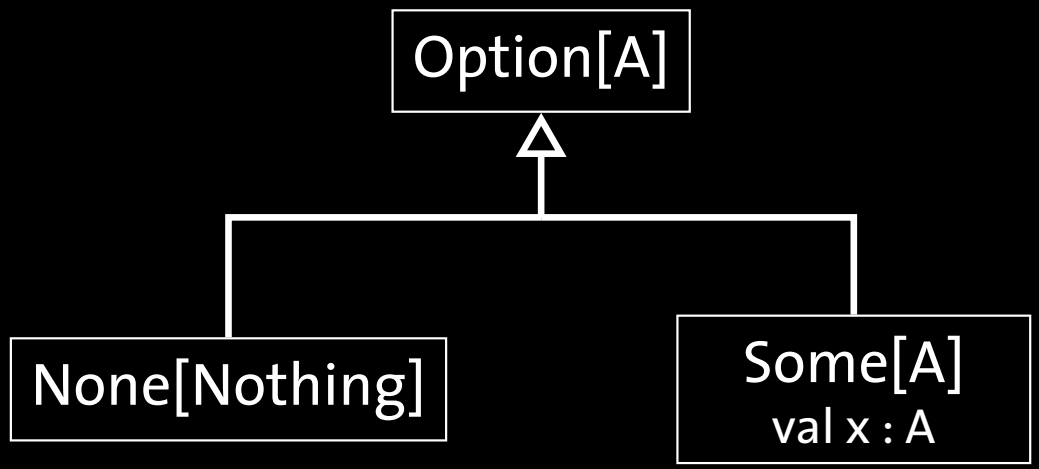
Retrieving Values from Maps

```
val toronto = capitals.get("Ontario") //Some("Toronto")
val unknown = capitals.get("123")   //None
```

Retrieving Values from Maps

```
val toronto = capitals.get("Ontario") //Some("Toronto")
val unknown = capitals.get("123")   //None
```

```
trait Map[A, B] {
  abstract def get(key : A) : Option[B]
  …
}
```



Pattern Matching on Objects

```
val retrieve = capitals.get("Nunavut")

retrieve match {
  case Some(x) => println("Found " + x)
  case None    => println("Unknown")
}

println(retrieve match {
  case Some(x) => "Found " + x
  case None    => "Unknown"
})
```

Pattern Matching on Objects

```
val retrieve = capitals.get("Nunavut")

retrieve match {
  case Some(x) => println("Found " + x)
  case None    => println("Unknown")
}

println(retrieve match {
  case Some(x) => "Found " + x
  case None    => "Unknown"
})
```

Pattern Matching on Objects

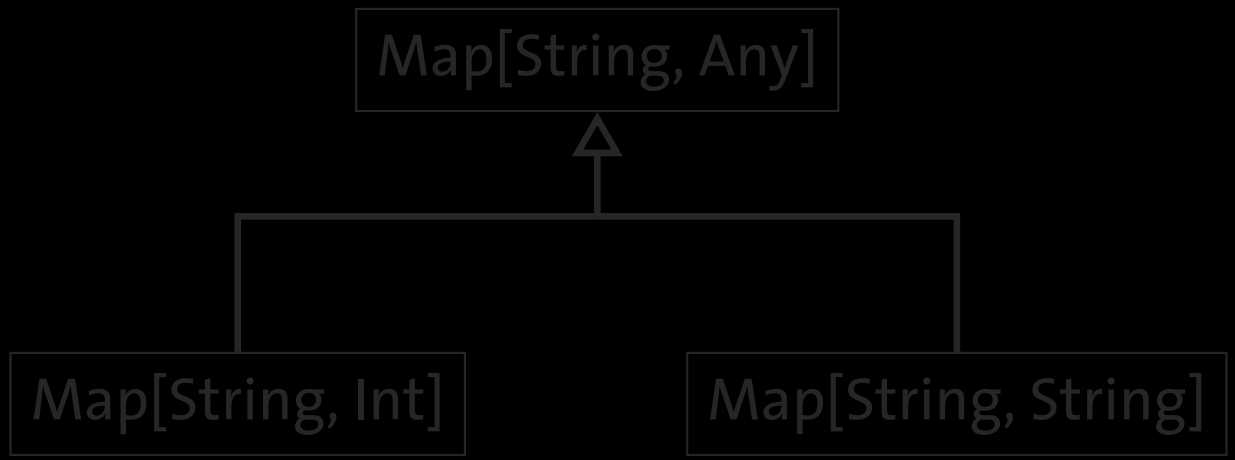
```
val retrieve = capitals.get("Nunavut")

retrieve match {
  case Some(x) => println("Found " + x)
  case None    => println("Unknown")
}

println(retrieve match {
  case Some(x) => "Found " + x
  case None    => "Unknown"
})
```

Co-Variance

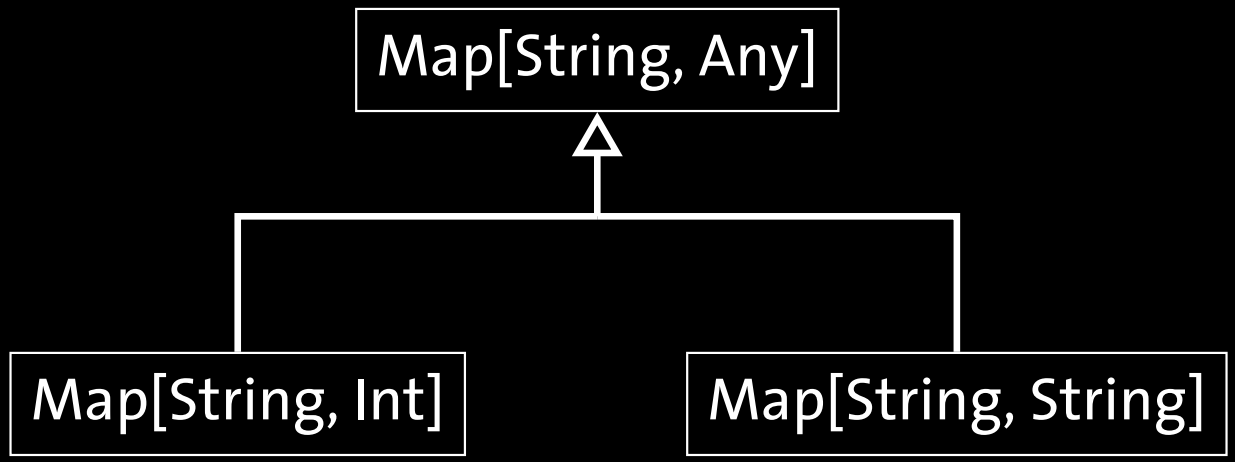
```
trait Map[A, +B]
```




```
val strMap = Map("Foo" -> "Bar")  //Map[String, String]
val intMap = Map("One" -> 1)      //Map[String, Int]
val mergedMap = strMap ++ intMap  //Map[String, Any]
```

Co-Variance

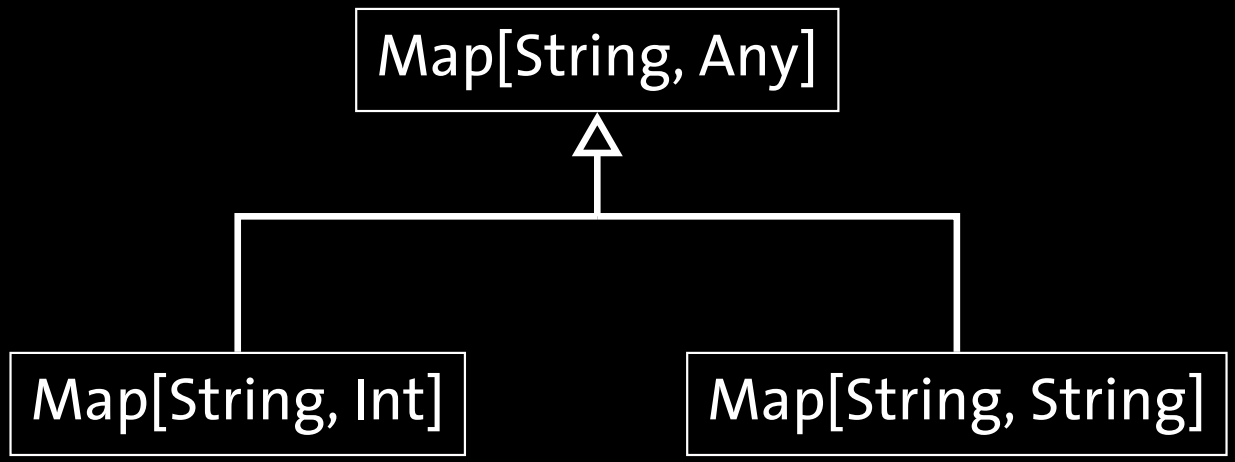
```
trait Map[A, +B]
```




```
val strMap = Map("Foo" -> "Bar")  //Map[String, String]
val intMap = Map("One" -> 1)      //Map[String, Int]
val mergedMap = strMap ++ intMap  //Map[String, Any]
```

Co-Variance

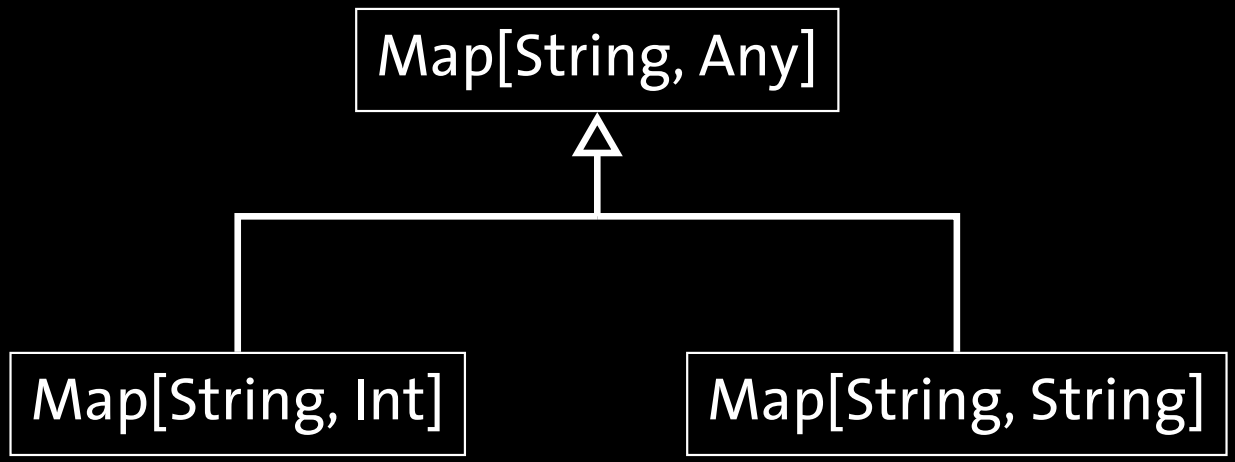
```
trait Map[A, +B]
```




```
val strMap = Map("Foo" -> "Bar")  //Map[String, String]
val intMap = Map("One" -> 1)      //Map[String, Int]
val mergedMap = strMap ++ intMap  //Map[String, Any]
```

Co-Variance

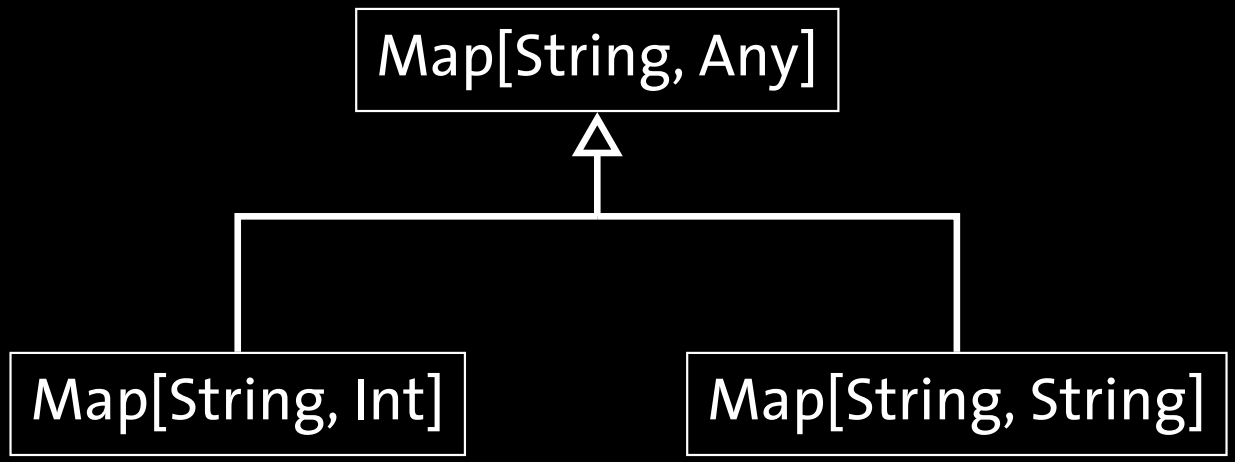
```
trait Map[A, +B]
```




```
val strMap = Map("Foo" -> "Bar")  //Map[String, String]
val intMap = Map("One" -> 1)      //Map[String, Int]
val mergedMap = strMap ++ intMap  //Map[String, Any]
```

Co-Variance

```
trait Map[A, +B]
```




```
val strMap = Map("Foo" -> "Bar")  //Map[String, String]
val intMap = Map("One" -> 1)      //Map[String, Int]
val mergedMap = strMap ++ intMap  //Map[String, Any]
```

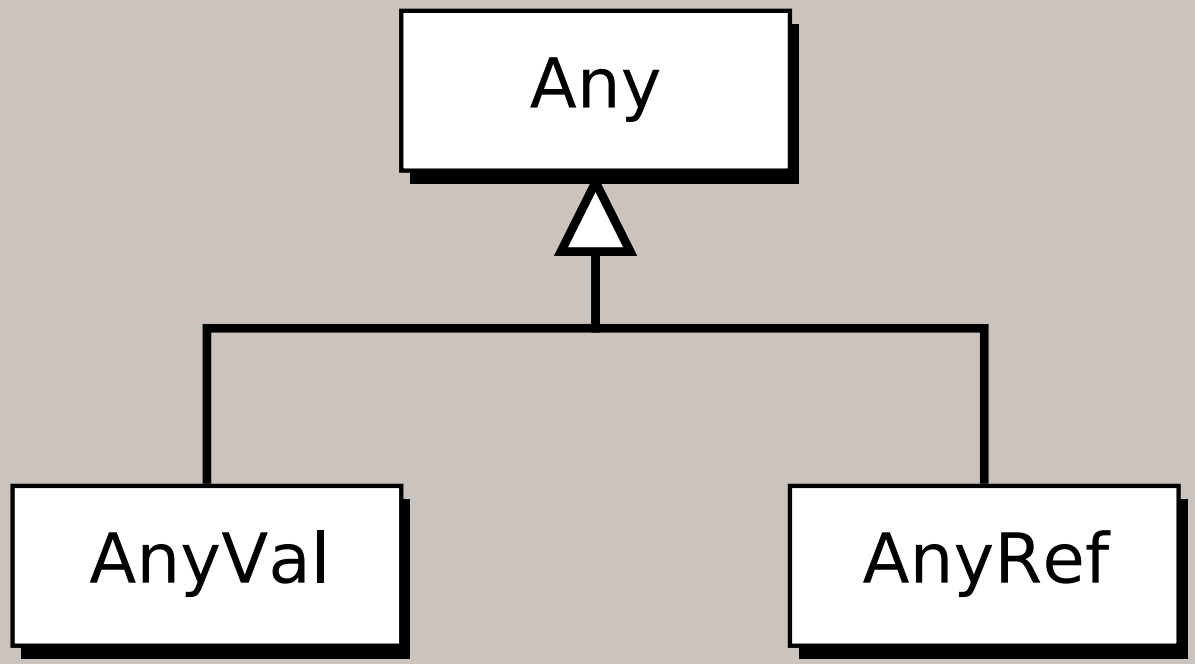
Associative Maps Conclusion

- **Factory pattern** built into language.
- Provide cleaner syntax through **operator methods**.
- Added methods to `String` using implicit conversion.
- **Co-variance** defines subtyping based on a type parameter.
- Get method returns Option which prevents NullPointerExceptions.
- Framework complexity is hidden from the user.

Outline

Scala's Type System

Everything is an object!

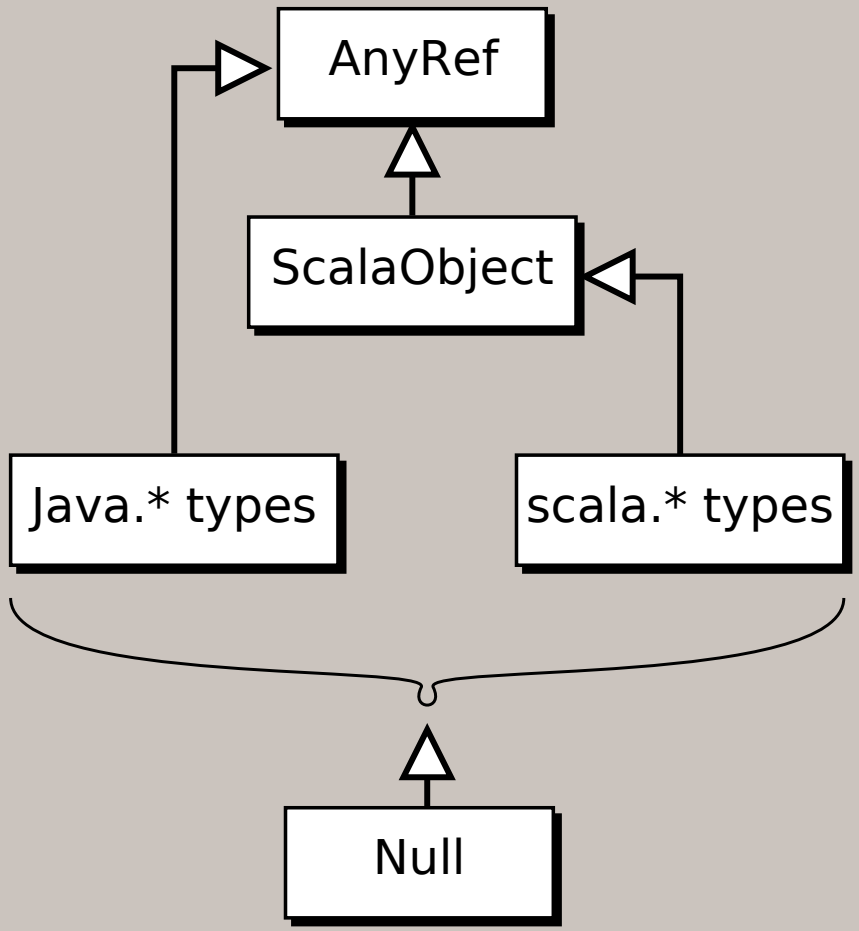


represents primitive types

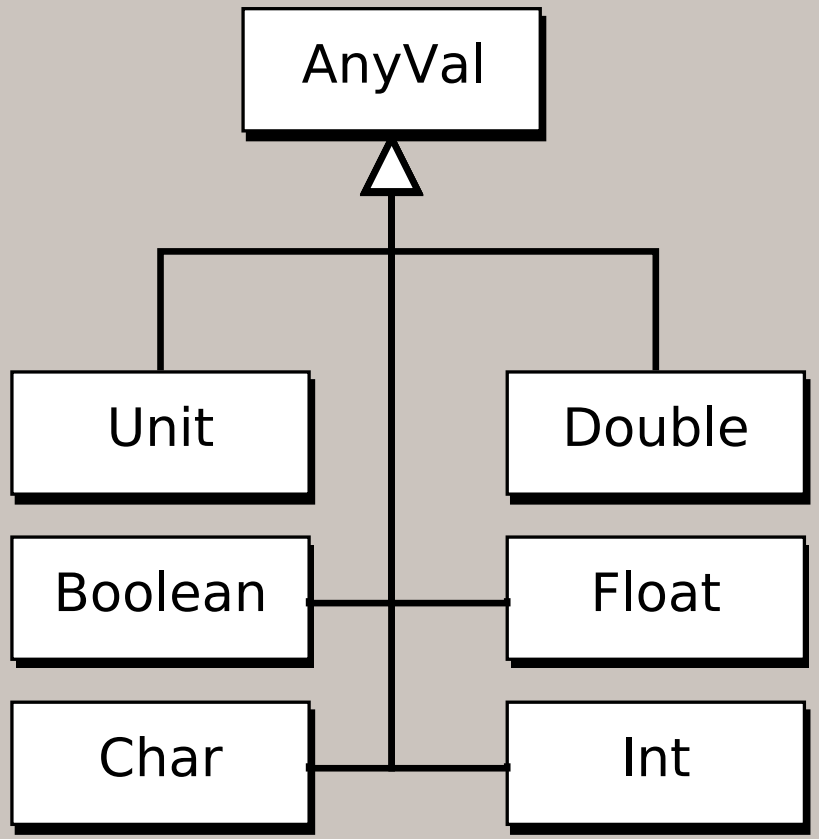
represents objects

[o] http://programming-scala.labs.oreilly.com/ch07.html##scalas-type-hierarchy

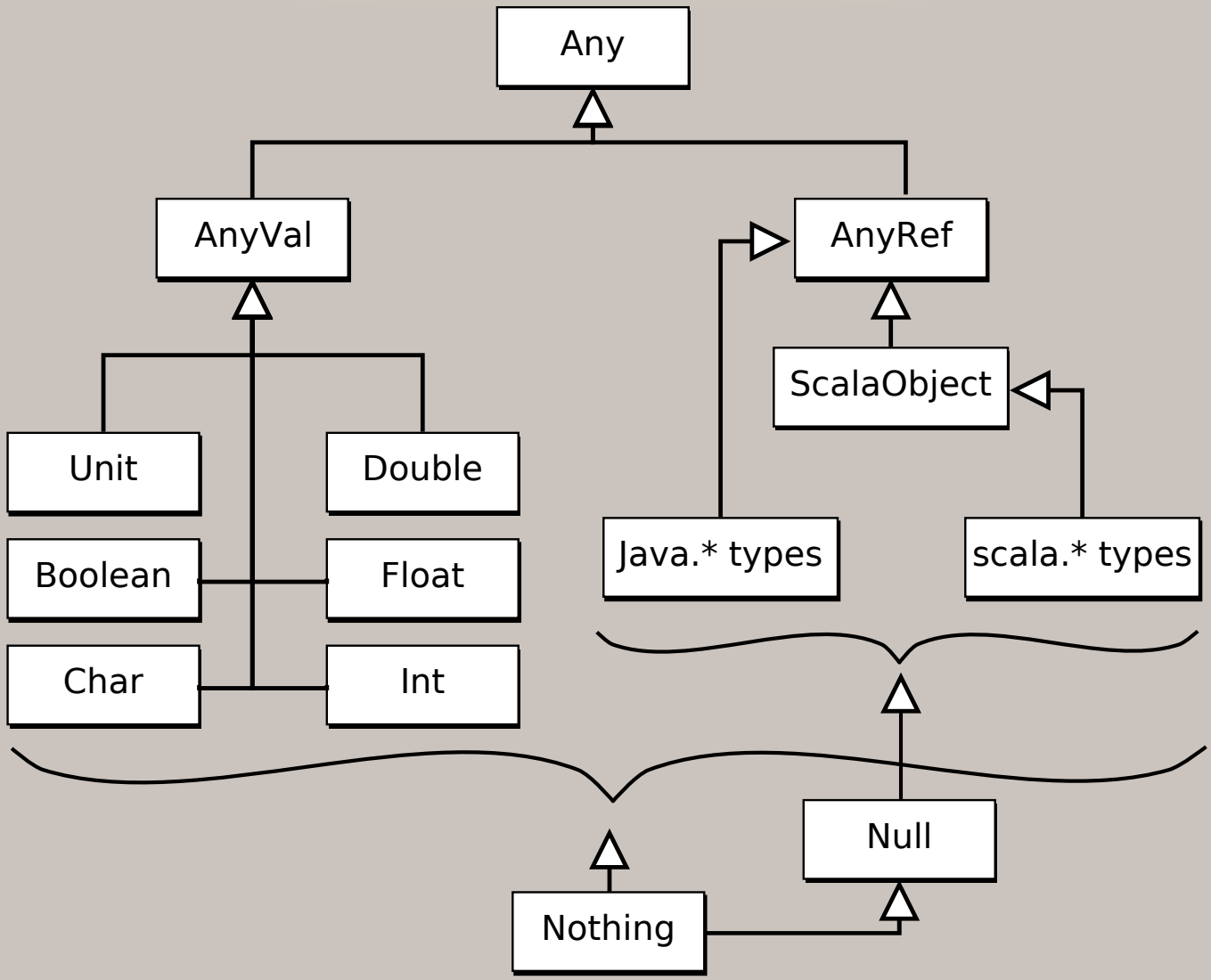
AnyRef Type Hierarchy



AnyVal Type Hierarchy



Combined Type Hierarchy



Outline

Baysick

```
object Lunar extends Baysick
  def main(args: Array[String]) = {
    10 PRINT "Welcome to Baysick Lunar Lander v0.9"
    20 LET ('dist := 100)
    30 LET ('v := 1)
    40 LET ('fuel := 1000)
    50 LET ('mass := 1000)

    //…

    100 PRINT "Distance " % 'dist % "km, " % "Velocity " …
    110 INPUT 'burn
    120 IF ABS('burn) <= 'fuel THEN 150
    130 PRINT "You don't have that much fuel"
    140 GOTO 100

    //…
```

[o] http://blog.fogus.me/2009/03/26/baysick-a-scala-dsl-implementing-basic/

Parser Combinators

BNF.

```
expr   ::= term ('+' term | '-' term)*
term   ::= factor ('*' factor | '/' factor)*
factor ::= floatingPointNumber | '(' expr ')'
```

Scala.

```
object ArithParser extends JavaTokenParsers {
  def expr : Parser[Any] = term ~ ("+" ~ term | "-" ~ term)*
  def term   = factor ~ ("*" ~ factor | "/" ~ factor)*
  def factor = floatingPointNumber | "(" ~ expr ~ ")"

  def parse(text : String) = parseAll(expr, text)
}
```

[o] http://www.ibm.com/developerworks/java/library/j-scala10248.html

Parser Combinators

BNF.

```
expr   ::= term ('+' term | '-' term)*
term   ::= factor ('*' factor | '/' factor)*
factor ::= floatingPointNumber | '(' expr ')'
```

Scala.

```
object ArithParser extends JavaTokenParsers {
  def expr : Parser[Any] = term ~ ("+" ~ term | "-" ~ term)*
  def term   = factor ~ ("*" ~ factor | "/" ~ factor)*
  def factor = floatingPointNumber | "(" ~ expr ~ ")"

  def parse(text : String) = parseAll(expr, text)
}
```

[o] http://www.ibm.com/developerworks/java/library/j-scala10248.html

Outline

Conclusions

- Static type system gives us type safety.
- Type inference can save a lot of typing.
- Everything is an object.
- Operators are just method calls.
- Implicit methods used ...
  - for automatic type conversion.
  - to introduce new methods.
- Pattern Matching.

Ambiguous implicit methods

```
implicit def any2ArrowAssoc[A](x: A)
  = new ArrowAssoc(x)

class ArrowAssoc[A](x: A) {
    def -> [B](y: B) = ...
}

implicit def any2MyArrow[A](x: A)
  = new MyArrow(x)

class MyArrow(x : Any) {
  def -> (y : MyClassB) = ...
}
```

Ambiguous implicit methods

```
implicit def any2ArrowAssoc[A](x: A)
  = new ArrowAssoc(x)

class ArrowAssoc[A](x: A) {
    def -> [B](y: B) = ...
}

implicit def any2MyArrow[A](x: A)
  = new MyArrow(x)

class MyArrow(x : Any) {
  def -> (y : MyClassB) = ...
}
```

Ambiguous implicit methods

```
implicit def any2ArrowAssoc[A](x: A)
  = new ArrowAssoc(x)

class ArrowAssoc[A](x: A) {
    def -> [B](y: B) = ...
}

implicit def any2MyArrow[A](x: A)
  = new MyArrow(x)

class MyArrow(x : Any) {
  def -> (y : MyClassB) = ...
}
```

Dangers of implicit conversions

```
"radar" == "radar".reverse ⇒ false
```

Bug in current version of Scala

comparing String and RichString using equals (==) returns false.

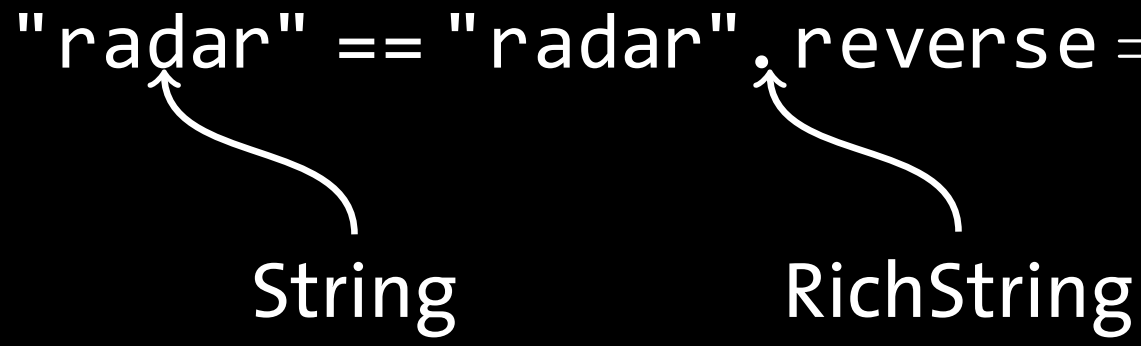
Conversion of types happens behind-your-back.

Dangers of implicit conversions

```
"radar" == "radar".reverse ⇒ false
```

- Bug in current version of Scala
  - comparing String and RichString using equals (==) returns false.
- Conversion of types happens behind-your-back.

Dangers of implicit conversions



- Bug in current version of Scala
  - comparing String and RichString using equals (==) returns false.
- Conversion of types happens behind-your-back.

Dangers of implicit conversions

- Bug in current version of Scala
  - comparing String and RichString using equals (==) returns false.
- Conversion of types happens behind-your-back.