

About Scala Types



The Scala Type System

- **Static, strong typing**
 - Type safety checked at compile time
 - Every value has a type
 - Compiler can infer types in many cases
- **Type model is based on Object Oriented principles**
 - Concrete types are classes
- **Types allow us to describe sets of values**
 - At different levels of abstraction
 - Expressed in different ways

Classes

- A class contains a set of properties
 - Data
 - Functions (methods)

```
class Person ( f: String, l: String, val age: Int = 18 ) {
  override def toString = s"$f $l ($age)"
}

val p = new Person ( "John", "Doe" )

println("The person is " + p) // The person is John Doe (18)
```

© J&G Services Ltd, 2016

Singleton Objects

- Class and a single instance of the class
 - Approximation to Singleton pattern
 - Use object keyword

```
object MessageObj {
  val hd = "Hello"
  val bd = "World"
  def showMessage = s"$hd $bd"
}

scala> MessageObj.hd
res34: String = Hello

scala> MessageObj.showMessage
res35: String = Hello World
```

© J&G Services Ltd, 2016

Companion Objects

- Object with same name as a class is a Companion Object
 - If part of the same compilation unit (ie same source file)
- Use to hold "static" members
- Access available to private properties of class
 - Can be used to implement factories

```
class Complex private ( val re: Int, val im: Int = 0 ) {
  ...
}
object Complex {
  def apply( r,i)  = new Complex(r, i)
}
```

val c1 = Complex(1, 2)

© J&G Services Ltd, 2016

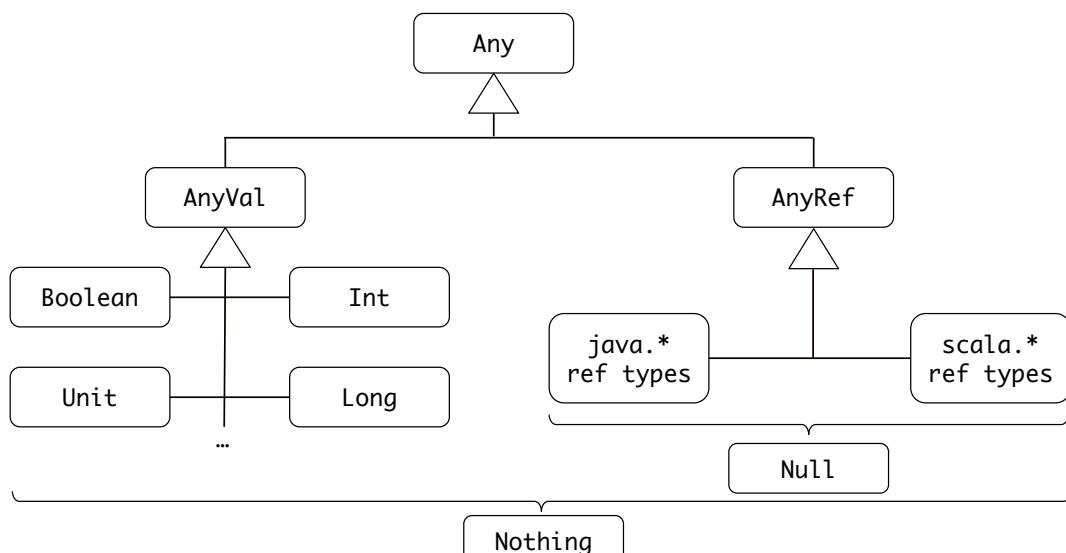
Case Classes

- Convenience mechanism for defining classes
 - Especially those exhibiting value semantics
- case class Money (dollars: Int, cents: Int)
- Compiler generates for commonly used methods
 - `toString`
 - `equals`
 - `copy / clone`
 - Companion object with `apply()` method for use as factory
- If no class parameters, use case object

case object SingleVal

© J&G Services Ltd, 2016

Scala Class Hierarchy



© J&G Services Ltd, 2016

Value Classes

- Classes extending `AnyVal`
 - Intended to wrap JVM primitive types
 - Compiler generates code that operates on unwrapped (primitive) values

```

scala> class MyStuff {
|   def doSomething ( i: Int ) = i.+ (4)
| }
defined class MyStuff

scala> new MyStuff().doSomething(3)
res6: Int = 7
  
```

```

scala> :javap -c MyStuff
Compiled from "<console>"
public class MyStuff {
  public int doSomething(int);
  Code:
    0: iload_1
    1: iconst_4
    2: iadd
    3: ireturn
  ...
}
  
```

© J&G Services Ltd, 2016

Value Classes

- Scala allows custom value classes to be defined
 - Builds in extra level of type safety
- Restrictions on new Value Classes:
 - Class must have exactly one parameter
 - Parameter must have public accessibility
 - Parameter must be val
 - No other vals allowed
 - No secondary constructors or initialisation statements
 - Must not be nested class

© J&G Services Ltd, 2016

Value Classes

- Example

```
scala> case class Mile ( m: Double ) extends AnyVal {
    |   def + ( other: Mile ) = Mile( this.m + other.m )
    |   def - ( other: Mile ) = Mile( this.m - other.m )
    |
}
defined class Mile

scala> val m1 = Mile (2.2)
m1: Mile = Mile(2.2)

scala> val m2 = Mile(4.2)
m2: Mile = Mile(4.2)

scala> m1 + m2
res8: Mile = Mile(6.4)
```

```
scala> m1 + 3.4
<console>:15: error: type mismatch;
 found   : Double(3.4)
 required: Mile
          m1 + 3.4
                      ^
scala> m1 + Mile(3.4)
res10: Mile = Mile(5.6)
```

A Mile can only be added to another Mile, not an arbitrary Double value

© J&G Services Ltd, 2016

Traits

- Abstract type representing properties of a type

```
trait HasId {
  def id: String
}
```

```
trait HasValue {
  def value: Double
}
```

- Properties are mixed in with class (and/or other traits)

```
class Stock ( val s_id: String ) extends HasId with HasValue {
  def id: String = s_id
  def value: Double = ???
}
```

```
class Bond( val b_id: String ) extends HasId with HasValue {
  def id: String = b_id
  def value: Double = ???
}
```

© J&G Services Ltd, 2016

Sealed Types

- Sealed types are types that can only be extended in the same compilation unit (source file)

- Normally abstract
- Allows control over subtypes
- Subtypes normally final
- Used to create Algebraic Sum Data Types

```
sealed trait Expression

final case class Const(v: Int) extends Expression
final case class Neg(e: Expression) extends Expression
final case class Add ( l: Expression, r: Expression ) extends Expression
```

$10 + (- (3 + 4))$

```
scala> val expr = Add ( Const(10), Neg ( Add( Const(3), Const(4) ) ) )
```

© J&G Services Ltd, 2016

Sealed Types

- Often used to define DSLs
 - Pattern Matching can be used to build an interpreter for the DSL

```
object ExpressionInterpreter {

  def eval ( e: Expression ): Int = e match {
    case Const(c) => c
    case Neg(e) => - eval(e)
    case Add(l, r) => eval(l) + eval(r)
  }
}

scala> val e1 = Add( Const(10), Neg( Add( Const(3), Const(4) ) ) )
e1: Add = Add(Const(10),Neg(Add(Const(3),Const(4))))  

scala> ExpressionInterpreter.eval(e1)
res13: Int = 3
```

© J&G Services Ltd, 2016

Sealed Types

- Sealed type hierarchy allows compiler to perform "exhaustiveness checking" in pattern match
 - Compiler knows all possible subtypes
 - Error not to include all possibilities in match
 - Alternative is MatchError exception at runtime

```
object ExpressionInterpreter {

  def eval ( e: Expression ): Int = e match {
    case Const(c) => c
    case Add(l, r) => eval(l) + eval(r)
  }
}
```

Match must include the
Neg case otherwise
MatchError exception
may occur

© J&G Services Ltd, 2016

Compound Types

- Represent intersections of types
 - Composed by mixing traits together

```
trait CanOpen { def open }
```

```
trait CanClose { def close }
```

```
class A extends CanOpen with CanClose {
  def open = println("A open")
  def close = println("A close")
}
```

```
class B extends CanOpen with CanClose {
  def open = println("B open")
  def close = println("B close")
}
```

```
def useIt ( it: CanOpen with CanClose ) = {
  it.open
  it.close
}
```

```
scala> useIt ( new A )
A open
A close
```

```
scala> useIt ( new B )
B open
B close
```

© J&G Services Ltd, 2016

Structural Typing

- Specify types by required properties
 - Static "duck typing"

```
def useIt2 ( it: { def open: Unit; def close: Unit } ) = {
  it.open
  it.close
}
```

```
scala> useIt2 ( new A )
A open
A close
```

```
scala> object OpenOnly { def open = println("OpenOnly") }
defined object OpenOnly
```

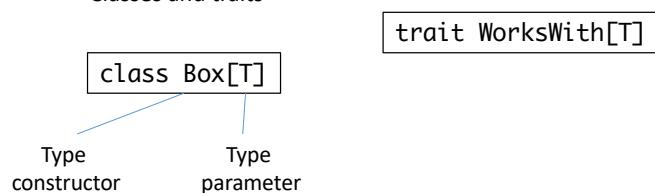
```
scala> useIt2 ( OpenOnly )
<console>:14: error: type mismatch;
 found   : OpenOnly.type
 required: AnyRef{def open: Unit; def close: Unit}
           useIt2 ( OpenOnly )
```

OpenOnly does not define close method as required

© J&G Services Ltd, 2016

Type Parameters

- Types can be defined using one or more parameters
 - Classes and traits



- Concrete types require parameters to be substituted

- May be inferred by the compiler or explicitly provided

- Examples

- List[T], Option[T], Map[K,V], Future[T], Try[T]

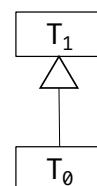
© J&G Services Ltd, 2016

Variance

- Describes the effect on parameterised types of inheritance

- Assume T_1 is a subtype of T_0

- What can we assume about (e.g.) List[T_1] and List[T_0]?



- Invariant

- No relationship

List[A]

- Covariant

- List[T_1] is a subtype of List[T_0]

List[+A]

- Contravariant

- List[T_1] is a supertype of List[T_0]

List[-A]

© J&G Services Ltd, 2016

Variance and Mutability

- Covariance implies read-only (immutable) type
 - Insertion of elements can break type safety
 - Collection types that offer covariance implement insert through defensive copying
 - E.g. List[T]
- Contravariance implies write-only types
 - Reading of elements can break type safety
 - E.g. Function types are contravariant in argument types, covariant in result type
- Invariance implies read and write

© J&G Services Ltd, 2016

Type Bounds

- Allow type parameters to be restricted according to inheritance hierarchy

```
scala> class Bag [ A <: Fruit ]
defined class Bag

scala> val b1 = new Bag [Banana]
b1: Bag[Banana] = Bag@38f981b6

scala> val b2 = new Bag [String]
<console>:13: error: type arguments [String] do not conform to
class Bag's type parameter bounds [A <: Fruit]
          val b2 = new Bag [String]
```

class Fruit
 class Apple extends Fruit
 class Banana extends Fruit

A must be a subtype of Fruit

© J&G Services Ltd, 2016

Type Bounds

- Allow type parameters to be restricted according to inheritance hierarchy

```
scala> class Bag [ A >: Fruit ]  
defined class Bag  
  
scala> val b1 = new Bag [Banana]  
<console>:14: error: type arguments [Banana] do not conform  
to class Bag's type parameter bounds [A >: Fruit]  
      val b1 = new Bag [Banana]  
           ^  
  
scala> val b1 = new Bag [AnyRef]  
b1: Bag[AnyRef] = Bag@d13960e
```

```
class Fruit  
class Apple extends Fruit  
class Banana extends Fruit
```

A must be a supertype of Fruit

© J&G Services Ltd, 2016

Type Parameters and Methods

- Methods can be parameterised by type
- Example: put method for covariant collection
 - Return copy with new element added
 - Need to infer type of new collection

```
class Bag[+A] { val stuff: Seq[A] = Seq() {  
  def get:A = stuff.head  
  def put [B >: A] (n: B) = new Bag[B](stuff :+ n)  
}  
  
val a = new Bag[Apple]( Seq( new Apple, new Apple ) )  
val a2 = a.put(new Apple) // OK, a2 is a Bag[Apple]  
val a3 = a.put(new Banana) // OK, but now a3 is a Bag[Fruit]
```

© J&G Services Ltd, 2016

Type Aliases

- Provide a name for a type

- Alternative name for class or trait
- Name for structural or compound type
- ...

- Improves code readability

```
scala> type ID = String
defined type alias ID

scala> type Openable = { def open }
defined type alias Openable

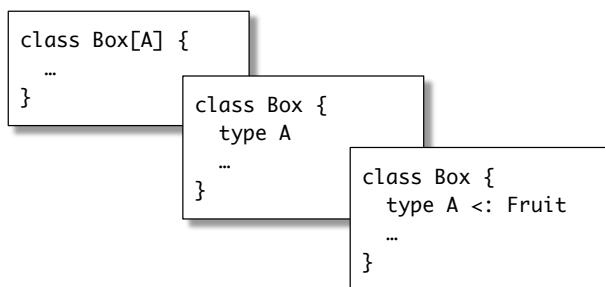
scala> type CanOpenAndClose = CanOpen with CanClose
defined type alias CanOpenAndClose

scala> def useIt ( it: CanOpenAndClose ) {
    |   it.open
    |   it.close
    | }
useIt: (it: CanOpenAndClose)Unit
```

© J&G Services Ltd, 2016

Type Members

- Types can be members of other types
 - Classes, traits, objects
- Type members can be abstract
 - Can provide alternative to type parameters in certain cases



© J&G Services Ltd, 2016

Nested Types

- Concrete types may be defined inside other types

- Class, trait or object

```
object OuterObj {
  class Inner
}

scala> val x: OuterObj.Inner = new OuterObj.Inner
x: OuterObj.Inner = OuterObj$Inner@3387ab0
```

- Types nested in object similar to Java static inner types

- Use import to simplify

```
scala> import OuterObj.-
import OuterObj.-

scala> val x = new Inner
x: OuterObj.Inner = OuterObj$Inner@342394b3
```

© J&G Services Ltd, 2016

Path Dependent Types

- Types defined within class are defined relative to *instance*

- Take care over type equivalence

```
scala> val o1 = new OuterClass
o1: OuterClass = OuterClass@38093ffe

scala> val o2 = new OuterClass
o2: OuterClass = OuterClass@3ba1f56e

scala> val oi1 = new o1.InnerClass
oi1: o1.InnerClass = OuterClass$InnerClass@1fd35a92

scala> val oi2 = new o2.InnerClass
oi2: o2.InnerClass = OuterClass$InnerClass@27b7204
```

```
class OuterClass {
  class InnerClass
}
```

These two objects
do not have the
same type

© J&G Services Ltd, 2016

Path Dependent Types

- Compiler uses type path to ensure type correctness

```
scala> def foo ( a: o1.InnerClass ) = a
foo: (a: o1.InnerClass)o1.InnerClass

scala> foo(oi1)
res32: o1.InnerClass = OuterClass$InnerClass@1fd35a92

scala> foo(oi2)
<console>:21: error: type mismatch;
 found   : o2.InnerClass
 required: o1.InnerClass
    foo(oi2)
           ^
```

© J&G Services Ltd, 2016

Path Dependent Types

- Use type projection to relax restriction if required

```
scala> def bar ( a: OuterClass#InnerClass ) = a
bar: (a: OuterClass#InnerClass)OuterClass#InnerClass

scala> bar(oi1)
res34: OuterClass#InnerClass = OuterClass$InnerClass@1fd35a92

scala> bar(oi2)
res35: OuterClass#InnerClass = OuterClass$InnerClass@27b7204
```

© J&G Services Ltd, 2016

Path Dependent Types

- Example
- Represent a board for playing games
 - Board coordinates are dependent on dimensions

```
case class Board( len: Int, height: Int ) {

  case class Coordinate ( x: Int, y: Int ) {
    require ( 0 <= x && x < len && 0 <= y && y < height )
  }

  val occupied = scala.collection.mutable.Set[Coordinate]()
}
```

© J&G Services Ltd, 2016

Path Dependent Types

```
scala> val b1 = Board(20, 20)
b1: Board = Board(20,20)

scala> val b2 = Board(30, 30)
b2: Board = Board(30,30)

scala> val c1 = b1.Coordinate(15, 15)
c1: b1.Coordinate = Coordinate(15,15)

scala> val c2 = b2.Coordinate(25, 25)
c2: b2.Coordinate = Coordinate(25,25)
```

```
scala> b1.occupied += c1
res36: b1.occupied.type =
Set(Coordinate(15,15))

scala> b2.occupied += c2
res37: b2.occupied.type =
Set(Coordinate(25,25))

scala> b1.occupied += c2
<console>:22: error: type mismatch;
 found   : b2.Coordinate
 required: b1.Coordinate
           b1.occupied += c2
                           ^

```

© J&G Services Ltd, 2016

Existential Types

- Suppose we wish to write a function as follows

```
def showSize ( s: Set[ ] ) = println(s.size)
```

What type should be specified for maximum generality?

- Scala demands a type parameter

- No raw types as in Java

© J&G Services Ltd, 2016

Existential Types

- Use type Any, as we don't care about the element type

```
def showSize ( s: Set[Any] ) = println(s.size)
```

- Compiles, but not generally useful

- Set type is invariant in its parameter
- Can only use with Set[Any]

```
scala> val si = Set(1,2,3)
si: scala.collection.immutable.Set[Int] = Set(1, 2, 3)

scala> showSize(si)
<console>:14: error: type mismatch;
 found   : scala.collection.immutable.Set[Int]
 required: Set[Any]
```

© J&G Services Ltd, 2016

Existential Types

- Introduce type parameter for element type

```
def showSize [T] ( s: Set[T] ) = println(s.size)
```

- Works, but clumsy

```
scala> val si = Set(1,2,3)
si: scala.collection.immutable.Set[Int] = Set(1, 2, 3)

scala> showSize(si)
3
```

- Method now carries type parameter for type of Set elements
 - Only interested in Set type

© J&G Services Ltd, 2016

Existential Types

- Use "Existential Type"

```
def showSize ( s: Set[T] forSome { type T } ) = println(s.size)
```

- Or, shorthand

```
def showSize ( s: Set[_] ) = println(s.size)
```

- Typesafe mechanism to abstract away from element type

```
scala> val si = Set(1,2,3)
si: scala.collection.immutable.Set[Int] = Set(1, 2, 3)

scala> showSize(si)
3
```

© J&G Services Ltd, 2016

Existential Types

- Existential type can have upper and/or lower bounds

```
def showSize ( s: Set[T] forSome { type T <: Fruit} ) = println(s.size)
def showSize ( s: Set[_ <: Fruit] ) = println(s.size)
```

```
scala> val sa = Set(new Apple, new Apple)
sa: scala.collection.immutable.Set[Apple] = Set(Apple@ce15aa, Apple@3dabf112)
scala> showSize(sa)
2
scala> val ss = Set("one", "two", "three")
ss: scala.collection.immutable.Set[String] = Set(one, two, three)
scala> showSize(ss)
<console>:15: error: type mismatch;
 found   : scala.collection.immutable.Set[String]
 required: Set[_ <: Fruit]
```

© J&G Services Ltd, 2016

Types and Kinds

- Int is a *type*

```
scala> :kind -v Int
scala.Int's kind is A
*
This is a proper type.
```

- Set[_] is a *type constructor*

- A description of how to build a type
- Supply parameter to build the type

```
scala> :kind -v Set[_]
scala.collection.immutable.Set's kind is F[A]
* -> *
This is a type constructor: a 1st-order-kinded type.
```

© J&G Services Ltd, 2016

Types and Kinds

- Container types may be nested

```
scala> trait Box[ F[_] ]  
defined trait Box
```

- Leads to Higher Kinded Types

- Similar in concept to higher order functions

```
scala> :kind -v Box[List]  
Box's kind is X[F[A]]  
(* -> *) -> *  
This is a type constructor that takes type constructor(s): a higher-kinded type.
```

```
scala> :kind -v Box[Option]  
Box's kind is X[F[A]]  
(* -> *) -> *  
This is a type constructor that takes type constructor(s): a higher-kinded type.
```

© J&G Services Ltd, 2016