Implicits and Typeclasses

Implicit

- Implicit definitions are used (inserted) by the compiler when necessary
  - Method definitions
  - Method parameters
  - Classes
- Multiple use cases
  - Transparent conversion between types
  - Flexible defaults for method parameters
  - Helping to define bounds for type parameters
  - Basis for type classes

Implicit Views – Transparent Type Conversion

- Example: String to Int

```
scala> math.max(3,5)
res39: Int = 5

scala> math.max(3,"5")
<console>:16: error: overloaded method value max with alternatives:
  (x: Double,y: Double)Double <and>
  (x: Float,y: Float)Float <and>
  (x: Long,y: Long)Long <and>
  (x: Int,y: Int)Int
 cannot be applied to (Int, String)
       math.max(3,"5")
            ^
scala> def strToInt(s: String):Int = s.toInt
strToInt: (s: String)Int

scala> math.max(3, strToInt("5") )
res41: Int = 5
```


Implicit Views – Transparent Type Conversion

- Make conversion method "implicit"
  - Happens automatically

```
scala> implicit def strToInt(s: String) = s.toInt
strToInt: (s: String)Int

scala> math.max(3,"5")
res44: Int = 5
```

- Compiler identifies function to use by signature
  - String => Int
  - Definition must be in scope
- If more than one function with this signature, compilation fails
  - Ambiguity

Resolving Implicit Definitions

- Implicit definitions must be visible to compiler
- Current Scope
  - Local definitions
  - Members of enclosing scope (class, package)
  - Imported identifiers
- Implicit Scope
  - Companion objects of associated types
  - Source and target type
  - Relevant type parameters
  - All parts of a compound type

Using Implicit Views

- Complex Number type

```
class Complex ( val re: Int = 0, val im: Int = 0 ) {
  def + ( that: Complex ) =
          new Complex( this.re + that.re, this.im + that.im )
  def - ( that: Complex ) =
          new Complex( this.re - that.re, this.im - that.im )
  override def toString() =
          "%d + %di".format(this.re, this.im)
}

object Complex {
  implicit def intToComplex ( i: Int) = new Complex(i)
}
```

```
scala> val c1: Complex = 3
c1: Complex = 3 + 0i

scala> c1 + 2
res45: Complex = 5 + 0i
```

- Implicit def in companion object
  - No import necessary to use it
  - Can be overridden locally if necessary

View Bounds

- Further means of qualifying type parameters
  - "type can be viewed as"
  - Relies on implicit view being in scope

Must be Implicit conversion from type A to Int

```
class Box[ A <% Int ] (val x: A) {
  def multBy3 = x * 3
}
```

```
scala> val b1 = new Box[Int](3)
b1: Box[Int] = Box@2c039ac6

scala> b1.multBy3
res0: Int = 9

scala> val b2 = new Box[String]("3")
<console>:12: error: No implicit view available from String => Int.
       val b2 = new Box[String]("3")
                ^
```


View Bounds

- Add implicit view

```
scala> implicit def strToInt( s: String ) = s.toInt
strToInt: (s: String)Int

scala> val b2 = new Box[String]("3")
b2: Box[String] = Box@20b2475a

scala> b2.multBy3
res1: Int = 9
```

```
class Box[ A <% Int ] (val x: A) {
  def multBy3 = x * 3
}
```

Adding Functionality to Types

- Without subtyping
  - "Pimp my library"
  - Define wrapper type to contain additional functions
  - Define implicit conversion from source class to the wrapper

```
class IntSquare ( val i: Int ) extends AnyVal {
  def square: Int = i * i
}


scala> implicit def intToIntSq ( i: Int ) = new IntSquare(i)
intToIntSq: (i: Int)IntSquare

scala> 4 square
res2: Int = 16
```


Adding Functionality to Types

- Package in object for easier use

```
object Utils {

  class IntSquare ( val i: Int ) extends AnyVal {
    def square: Int = i * i
  }

  object IntSquare {
    implicit def intToIntSquare ( n: Int ) : IntSquare = new IntSquare(n)
  }

}
```

```
scala> import Utils._
import Utils._

scala> 4 square
res51: Int = 16
```

Adding Functionality to Types

- Implicit class combines two stages
  - Available since Scala 2.10

```
implicit class IntOps ( i: Int ) {
  def squared: Int = i * i
  def cubed: Int = i * i * i
}


scala> 3 squared
res52: Int = 9

scala> 4 cubed
res53: Int = 64
```


Implicit Parameters

- Method/function parameters can be defined as implicit
  - Allows flexible approach to default values
  - Only allowed in last parameter list (see curried functions)

```
scala> def power ( a: Int )  ( implicit b: Int ) = math.pow(a,b)
power: (a: Int)(implicit b: Int)Double

scala> power(2)(3)
res54: Double = 8.0

scala> power(3)
<console>:33: error: could not find implicit value for parameter b: Int
       power(3)
             ^
```

```
scala> implicit val exponent: Int = 2
exponent: Int = 2

scala> power(3)
res56: Double = 9.0
```

Implicit Parameters

- Resolution of implicit arguments is done as for implicit conversions
  - Based on type
  - Can be val or def
  - Same scoping rules
- Can be mixed with default parameter values
  - Not advised, can be misleading
- Provides a mechanism for caller-defined default values
  - Rather than implementer-defined default

Implicit Parameters

- Executing task in concurrent context

```
import java.util.concurrent._

def doTask ( r: Runnable ) ( implicit e: Executor ) =
  e.execute(r)
```

```
scala> implicit val ex = Executors.newFixedThreadPool(5)
ex: java.util.concurrent.ExecutorService =
                        java.util.concurrent.ThreadPoolExecutor@1ca2d595 ….

scala> val task = new Runnable { override def run = println("Hello") }
task: Runnable = $anon$1@3ee39a1c

scala> doTask(task)
Hello
```

Execute task in whatever threading context has been set in implicit scope

About Type Classes

- "Ad hoc" polymorphism
  - Allows new functionality to be added to existing types
  - More powerful than implicit views/classes
- Based on ideas from Haskell
- Implementation possible in Scala
  - Uses parameterised traits
  - Implicit
- Very common and powerful pattern
  - Support integrated into type bounding mechanism

Why Type Classes?

- Consider the following classes

```
class Person ( fName: String, lName: String, val age: Int ) {
  val name = s"${fName} ${lName}"
  override def toString = s"${name}: ${age}"
}

scala> val gb = new Person("George", "Ball", 21)
gb: Person = George Ball: 21
```

```
class Trade ( val id: String, val side: String, val sym: String,
              val amount: Int, val unitPrice: Double ) {
  override def toString =
        s"${if ( side == "b" ) "Buy" else "Sell"} ${amount} of \
                                                ${sym} at ${unitPrice}"
}

scala> val t1 = new Trade("T1", "b", "AAPL", 1000, 105.0)
t1: Trade = Buy 1000 of AAPL at 105.0
```

Why Type Classes?

- Requirement is to serialise to XML

```
class Person ( fName: String, lName: String, val age: Int ) {
  val name = s"${fName} ${lName}"
  override def toString = s"${name}: ${age}"
  def toXML: scala.xml.Elem = <person>
      <name>{this.name}</name>
      <age>{this.age}</age>
    </person>
}
```

```
scala> val gb = new Person("George", "Ball", 21)
gb: Person = George Ball: 21

scala> gb toXML
res58: scala.xml.Elem =
<person>
      <name>George Ball</name>
      <age>21</age>
    </person>
```


Why Type Classes?

- Requirement is to serialise to XML

```
class Trade ( val id: String, val side: String, val sym: String,
              val amount: Int, val unitPrice: Double ) {
  …
  def toXML = <trade>
    <id>{this.id}</id>
    <side>{if (side == "b") "Buy" else "Sell"}</side>
    <sym>{this.sym}</sym>
    <amount>{this.amount}</amount>
    <unitPrice>{this.unitPrice}</unitPrice>
  </trade>
}
```

```
scala> t1 toXML
res61: scala.xml.Elem =
<trade>
    <id>T1</id>
    …
    <unitPrice>105.0</unitPrice>
  </trade>
```

Using Type Class

- Encapsulate required behaviour as a type
  - Normally a parameterised trait
  - This is the Type Class

```
trait XMLSerializer[A] {
  def toXML(a: A): scala.xml.Elem
}
```

- Create instance of the trait to define concrete behaviour for target type(s)
  - In implicit scope

```
implicit val personXML = new XMLSerializer[Person] {
  def toXML(p: Person) = <person><name>{p.name}</name>
                                <age>{p.age}</age></person>
}
```


Using Type Class

- Encapsulate required behaviour as a type
  - Normally a parameterised trait
  - This is the Type Class

```
trait XMLSerializer[A] {
  def toXML(a: A): scala.xml.Elem
}
```

- Create instance of the trait to define concrete behaviour for target type(s)
  - In implicit scope

```
implicit val tradeXML = new XMLSerializer[Trade] {
  def toXML(t: Trade) = <trade> <id>{t.id}</id>
                           <side>{if (t.side == "b") "Buy" else "Sell"}</side>
                           ...
                        </trade>
}
```

Using Type Class

- Use implicit class to encapsulate transformation functionality
  - Type class is implicit parameter to transform function

```
implicit class Serializer[A](val a: A) {
  def asXML(implicit instance: XMLSerializer[A]): scala.xml.Elem =
                                              instance toXML a
}
```

- Now functionality available on selected types
  - As if it were part of the type

```
scala> gb asXML
res62: scala.xml.Elem =
<person><name>George Ball</name>
        <age>21</age></person>
```

```
scala> t1 asXML
res63: scala.xml.Elem =
<trade> <id>T1</id>
    <side>Buy</side>
    <sym>AAPL</sym> <amount>1000</amount>
    <unitPrice>105.0</unitPrice>
  </trade>
```


Using Type Class

- Other types can have the functionality "added"
  - Define type class instance for the type in implicit scope

```
implicit val stringXML = new XMLSerializer[String] {
  def toXML(s: String): scala.xml.Elem = <str>{s}</str>
}

implicit val intXML = new XMLSerializer [Int] {
  def toXML(i: Int): scala.xml.Elem = <val>{i}</val>
}
```

```
scala> 4 asXML
res64: scala.xml.Elem = <val>4</val>

scala> "Foobar" asXML
res66: scala.xml.Elem = <str>Foobar</str>
```

Context Bounds for Types

- Improvement over View Bounds
  - Requires presence of a type class instance for the specified type

```
scala> def serializeToXML[A: XMLSerializer] (a: A) = a asXML
serializeToXML: [A](a: A)(implicit evidence$1: XMLSerializer[A])scala.xml.Elem
```

- Argument to serializeToXML must be of a type that has a type class instance defined

```
scala> serializeToXML(t1)
res67: scala.xml.Elem =
<trade> <id>T1</id>
    <side>Buy</side>
    <sym>AAPL</sym>
    <amount>1000</amount>
    <unitPrice>105.0</unitPrice>
  </trade>
```

```
scala> serializeToXML(4)
res68: scala.xml.Elem = <val>4</val>

scala> serializeToXML(new java.util.Date)
<console>:49: error: could not find implicit
value for evidence parameter of type
XMLSerializer[java.util.Date]
       serializeToXML(new java.util.Date)
                      ^
```


Doing Without the Implicit Class

- Type class instance can be accessed without an implicit class
  - Use Predef.implicitly method

```
def serializeToXML[A: XMLSerializer] (a: A) =
               implicitly[XMLSerializer[A]].toXML(a)
```

Does not compile unless type class instance in implicit scope

```
scala> serializeToXML(gb)
res71: scala.xml.Elem =
<person><name>George Ball</name><age>21</age></person>

scala> serializeToXML(false)
<console>:49: error: could not find implicit value for evidence parameter of type
XMLSerializer[Boolean]
       serializeToXML(false)
                      ^
```