# XML PROCESSING WITH SCALA AND YAIDOM

Yaidom: a Scala XML query and transformation API (Apache 2.0 license)

Showing yaidom by examples using XBRL

Created by chris.de.vreeze@ebpi.nl

Powered by reveal.js

# OVERVIEW OF THE PRESENTATION

- What is yaidom?
- Use case: XBRL
- Introducing Scala higher-order functions
- Introducing yaidom higher-order functions

---

- Namespace validation example
- XBRL context validation example
- XBRL context validation example, revisited
- Takeaway points about yaidom

# WHAT IS YAIDOM?

- An (open source) *XML query* and transformation API
- Leverages *Scala* and the Scala Collections API
- Defines some core concepts (ENames, QNames, Scope etc.)
- Its namespace support is built on these concepts
- Its XML query API is built on its namespace support
- The *same* query API is offered by *multiple* element implementations (why? e.g. XML diff vs. XML editor)
- Including your own custom ones (easy to add)
- Including type-safe ones for specific XML dialects (e.g. XBRL)

# USE CASE: XBRL

- Yaidom is shown using the XBRL example below
- XBRL is an XML-based (financial) reporting standard
- It is very XML-intensive
- A business report in XBRL is called an *XBRL instance*
- It reports *facts*
- Having *contexts* ("who", "when" etc.)
- And possibly *units* ("which currency", etc.)

```
<xbrli:xbrl xmlns:xbrli="http://www.xbrl.org/2003/instance"
  xmlns:cc2-i="cc2i" xmlns:cc-t="cct" xmlns:cd="nlcd" xmlns:iso4217="iso4217">
  <xbrli:context id="FY14d">
    <xbrli:entity>
      <xbrli:identifier scheme="http://www.cc.eu/cc-id">30267975
      </xbrli:identifier>
    </xbrli:entity>
    <xbrli:period>
      <xbrli:startDate>2014-01-01</xbrli:startDate>
      <xbrli:endDate>2014-12-31</xbrli:endDate>
    </xbrli:period>
  </xbrli:context>
  <xbrli:unit id="EUR">
    <xbrli:measure>iso4217:EUR</xbrli:measure>
  </xbrli:unit>
  <cc2-i:Equity contextRef="FY14d" unitRef="EUR"
    decimals="INF">95000</cc2-i:Equity>
  <cc-t:EntityAddressPresentation>
    <cd:POBoxNumber contextRef="FY14d">2312</cd:POBoxNumber>
    <cd:PostalCodeNL contextRef="FY14d">2501CD</cd:PostalCodeNL>
    <cd:PlaceOfResidenceNL contextRef="FY14d">Den Haag
    </cd:PlaceOfResidenceNL>
    <cd:CountryName contextRef="FY14d">Nederland</cd:CountryName>
  </cc-t:EntityAddressPresentation>
</xbrli:xbrl>
```

# INTRODUCING SCALA HIGHER-ORDER FUNCTIONS

- Scala has a rich Collections API
- The most commonly used collections are immutable
- Typically, collections are created from other collections by applying ("for-each-like") higher-order functions
- For example, function *filter* takes an element predicate, and keeps only those elements for which the predicate holds
- And method *map* takes a function, and replaces all elements by the result of applying the function

## First some yaidom basics:

- Method *findAllChildElems* finds all child elements
- *EName* stands for "expanded name"

## Below methods "filter" and "map" are shown:

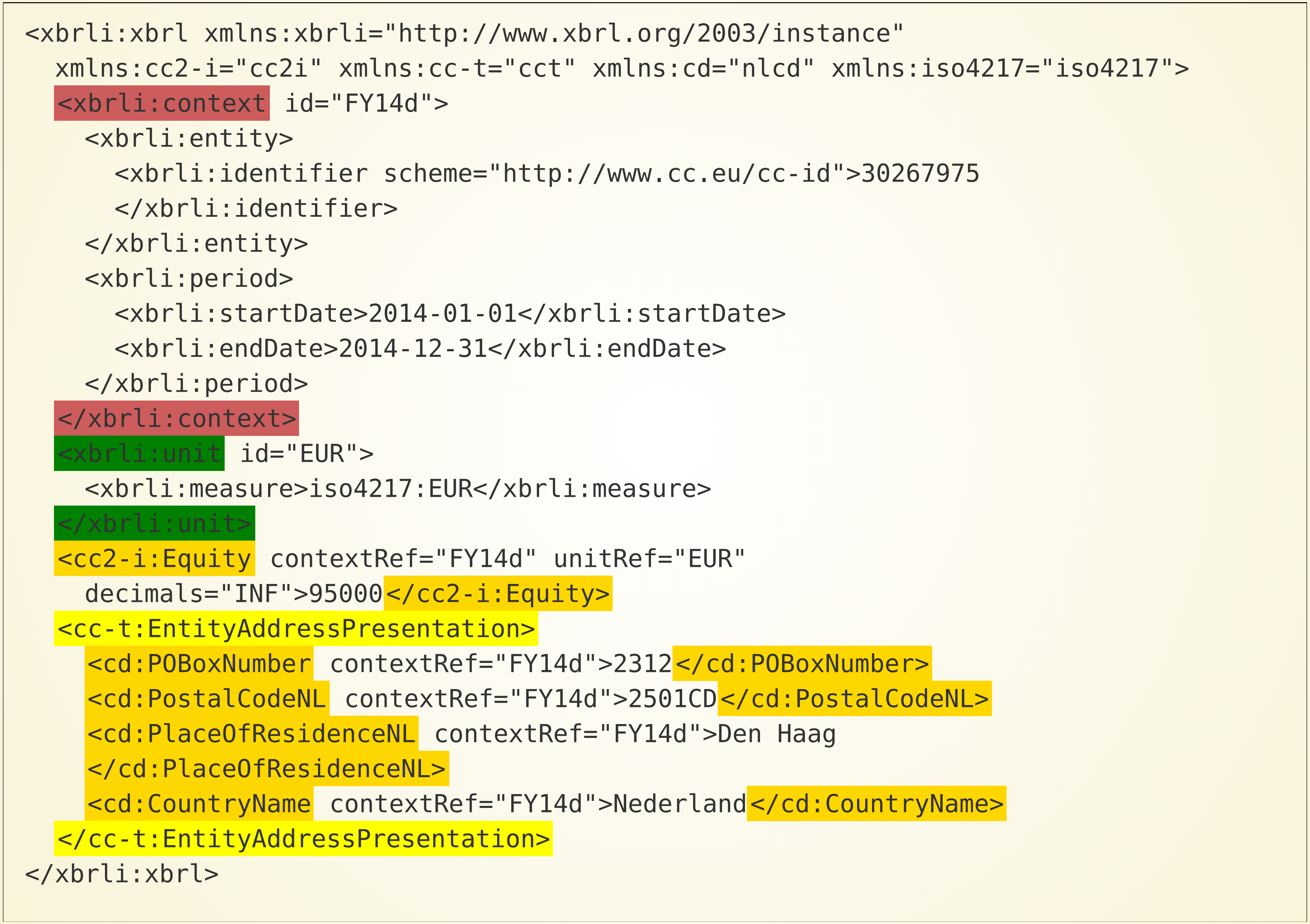
```
val xbrliNs = "http://www.xbrl.org/2003/instance"

val contexts =
  instance.findAllChildElems.filter(e =>
    e.resolvedName == EName(xbrliNs, "context"))

val contextIds =
  contexts.map(e => e.attribute(EName("id")))
```

# INTRODUCING YAIDOM HIGHER-ORDER FUNCTIONS

- Yaidom's query API offers many higher-order element methods that take an element predicate
- Most of these functions return a collection of elements
- E.g., method *filterChildElems* filters child elements
- Method *filterElems* filters descendant elements
- And method *filterElemsOrSelf* filters descendant-or-self elements
- They are somewhat similar to XPath axes, but return only elements
- If you understand these filtering methods, you understand them all
- Let's use them to find contexts, units and facts



```
<xbrli:xbrl xmlns:xbrli="http://www.xbrl.org/2003/instance"
  xmlns:cc2-i="cc2i" xmlns:cc-t="cct" xmlns:cd="nlcd" xmlns:iso4217="iso4217">
  <xbrli:context id="FY14d">
    <xbrli:entity>
      <xbrli:identifier scheme="http://www.cc.eu/cc-id">30267975
      </xbrli:identifier>
    </xbrli:entity>
    <xbrli:period>
      <xbrli:startDate>2014-01-01</xbrli:startDate>
      <xbrli:endDate>2014-12-31</xbrli:endDate>
    </xbrli:period>
  </xbrli:context>
  <xbrli:unit id="EUR">
    <xbrli:measure>iso4217:EUR</xbrli:measure>
  </xbrli:unit>
  <cc2-i:Equity contextRef="FY14d" unitRef="EUR"
    decimals="INF">95000</cc2-i:Equity>
  <cc-t:EntityAddressPresentation>
    <cd:POBoxNumber contextRef="FY14d">2312</cd:POBoxNumber>
    <cd:PostalCodeNL contextRef="FY14d">2501CD</cd:PostalCodeNL>
    <cd:PlaceOfResidenceNL contextRef="FY14d">Den Haag
    </cd:PlaceOfResidenceNL>
    <cd:CountryName contextRef="FY14d">Nederland</cd:CountryName>
  </cc-t:EntityAddressPresentation>
</xbrli:xbrl>
```

# Finding facts, contexts and units (as plain XML elements), regardless of the element implementation:

```
val ns = "http://www.xbrl.org/2003/instance"
val linkNs = "http://www.xbrl.org/2003/linkbase"

def hasCustomNs(e: Elem): Boolean = {
  !Set(Option(ns), Option(linkNs)).contains(
    e.resolvedName.namespaceUriOption)
}

val contexts = xbrlInstance.filterChildElems(withEName(ns, "context"))
val units = xbrlInstance.filterChildElems(withEName(ns, "unit"))
val topLevelFacts =
  xbrlInstance.filterChildElems(e => hasCustomNs(e))
val nestedFacts =
  topLevelFacts.flatMap(_.filterElems(e => hasCustomNs(e)))
val allFacts =
  topLevelFacts.flatMap(_.filterElemsOrSelf(e => hasCustomNs(e)))
```

# Non-trivial queries combine facts with their contexts and units:

```
val contextsById =
  contexts.groupBy(_.attribute(EName("id")))
val unitsById =
  units.groupBy(_.attribute(EName("id")))

// Use these Maps to look up contexts and units from
// (item) facts, with predictable performance ...
```

# NAMESPACE VALIDATION EXAMPLE

- To illustrate (low level) validations, let's check the use of "standard" namespaces
- In particular, let's validate rule 2.1.5 of the international FRIS standard
- The rule states that some commonly used namespaces should use their "preferred" prefixes in XBRL instances
- We also check the reverse, namely that those prefixes map to the expected namespaces
- For simplicity, assume that all namespace declarations are only in the root element

```
<xbrli:xbrl xmlns:xbrli="http://www.xbrl.org/2003/instance"
  xmlns:cc2-i="cc2i" xmlns:cc-t="cct" xmlns:cd="nlcd" xmlns:iso4217="iso4217">
  <xbrli:context id="FY14d">
    <xbrli:entity>
      <xbrli:identifier scheme="http://www.cc.eu/cc-id">30267975
      </xbrli:identifier>
    </xbrli:entity>
    <xbrli:period>
      <xbrli:startDate>2014-01-01</xbrli:startDate>
      <xbrli:endDate>2014-12-31</xbrli:endDate>
    </xbrli:period>
  </xbrli:context>
  <xbrli:unit id="EUR">
    <xbrli:measure>iso4217:EUR</xbrli:measure>
  </xbrli:unit>
  <cc2-i:Equity contextRef="FY14d" unitRef="EUR"
    decimals="INF">95000</cc2-i:Equity>
  <cc-t:EntityAddressPresentation>
    <cd:POBoxNumber contextRef="FY14d">2312</cd:POBoxNumber>
    <cd:PostalCodeNL contextRef="FY14d">2501CD</cd:PostalCodeNL>
    <cd:PlaceOfResidenceNL contextRef="FY14d">Den Haag
    </cd:PlaceOfResidenceNL>
    <cd:CountryName contextRef="FY14d">Nederland</cd:CountryName>
  </cc-t:EntityAddressPresentation>
</xbrli:xbrl>
```

```scala
// All namespace declarations must be in the root element

require(
  xbrlInstance.findAllElems.forall(_.scope == xbrlInstance.scope))
```

```scala
val standardScope = Scope.from(
  "xbrli" -> "http://www.xbrl.org/2003/instance",
  "xlink" -> "http://www.w3.org/1999/xlink",
  "link" -> "http://www.xbrl.org/2003/linkbase",
  "xsi" -> "http://www.w3.org/2001/XMLSchema-instance",
  "iso4217" -> "http://www.xbrl.org/2003/iso4217")

val standardPrefixes = standardScope.keySet
val standardNamespaceUris = standardScope.inverse.keySet
```

```scala
val subscope = xbrlInstance.scope.withoutDefaultNamespace filter {
  case (pref, ns) =>
    standardPrefixes.contains(pref) ||
      standardNamespaceUris.contains(ns)
}
require(subscope.subScopeOf(standardScope)) // fails on iso4217
```

# XBRL CONTEXT VALIDATION EXAMPLE

- Let's now validate rule 2.4.2 of the international FRIS standard
- The rule states that all contexts must be used
- We also check the reverse, that all context references indeed refer to existing contexts
- N.B. The latter check belongs to XBRL instance validation, not to FRIS validation for XBRL-valid instances

```
val ns = "http://www.xbrl.org/2003/instance"
val linkNs = "http://www.xbrl.org/2003/linkbase"

def hasCustomNs(e: Elem): Boolean = {
  !Set(Option(ns), Option(linkNs)).contains(
    e.resolvedName.namespaceUriOption)
}

val contexts = xbrlInstance.filterChildElems(withEName(ns, "context"))
val units = xbrlInstance.filterChildElems(withEName(ns, "unit"))
val topLevelFacts =
  xbrlInstance.filterChildElems(e => hasCustomNs(e))
val allFacts =
  topLevelFacts.flatMap(_.filterElemsOrSelf(e => hasCustomNs(e)))
```

```
val contextIds =
  contexts.map(_.attribute(EName("id"))).toSet

val usedContextIds =
  allFacts.flatMap(_.attributeOption(EName("contextRef"))).toSet

require(usedContextIds.subsetOf(contextIds))
require(contextIds.subsetOf(usedContextIds))
```

# XBRL CONTEXT VALIDATION EXAMPLE, REVISITED

- Let's hint at how to implement the same rule at a higher level of abstraction
- Yaidom makes it easy to support XML dialects by gradually adding types to XML elements
- So these custom yaidom elements offer the same yaidom query API, plus more
- This is different from O-X mappers, such as JAXB
- Let's assume such custom elements modeling XBRL instances and their components
- Then the validation code shown earlier could be reduced to something like the code shown below

# Expressive and type-safe validation code, using an imaginary yaidom extension for XBRL instances:

```
val contextIds = xbrlInstance.allContextsById.keySet

val usedContextIds =
  xbrlInstance.findAllItems.map(_.contextRef).toSet

require(usedContextIds.subsetOf(contextIds))
require(contextIds.subsetOf(usedContextIds))
```

# TAKEAWAY POINTS ABOUT YAIDOM

- Like the standard Scala XML library, yaidom leverages Scala and its Collections API
- Yet yaidom offers *multiple* element implementations behind the *same query API*
- Including James Clark's labeled element tree abstraction
- Or Saxon NodeInfo wrappers
- Or type-safe custom yaidom elements for XBRL data
- At EBPI, we use Scala and yaidom (as well as Saxon-EE) in our XBRL tooling
- Using Scala, we can stack layers of abstraction (XML, XLink, XBRL etc.)
- Results: high data quality and a quick time to market

Yaidom (Apache 2.0 license) can be found at
https://github.com/dvreeze/yaidom

chris.de.vreeze@ebpi.nl