Xtend Refcard

Hello World

```
1. class HelloWorld {
2.   def static void main(String[] args) {
3.     println("Hello World")
4.   }
5. }
```

Package Declaration

Package declarations can look like those in Java. Two small, optional differences:

An identifier can be escaped with a ^ character in case it conflicts with a keyword.

The terminating semicolon is optional.

```
1. package com.acme
```

Imports

Names are same as Java.

Again one can escape any names conflicting with keywords using a ^.

Terminating semicolon is optional.

Xtend also features static imports but allows only a wildcard * at the end, i.e. you cannot import single members using a static import. Non-static wildcard imports are deprecated for the benefit of better usability and well defined dependencies.

As in Java all classes from the java.lang package are implicitly imported.

```
1. import java.math.BigDecimal
2. import static java.util.Collections.*
```

Constructors

use the keyword new to declare a constructor. Constructors can also delegate to other constructors using this(args...) in their first line.

```
1. class MyClass extends AnotherClass {
2.   new(String s) {
3.     super(s)
4.   }
5.
6.   new() {
7.     this("default")
8.   }
9. }
```

The same rules with regard to inheritance apply as in Java.

The default visibility of constructors is public but you can also specify an explicit visibility public, protected, package or private.

Dispatch Methods

Method resolution and binding is done statically at compile time. Method calls are bound based on the static types of arguments. A dispatch method is declared using the keyword dispatch.

```
1. def dispatch printType(Number x) {
2.   "it's a number"
3. }
4.
5. def dispatch printType(Integer x) {
6.   "it's an int"
7. }
```

Methods

```
1. def String first(List<String> elements) {
2.   elements.get(0)
3. }
```

Start with the keyword def.

Xtend supports the static and can infer the return type if it is not explicitly given:

```
1. def static createInstance() {
2.   new MyClass('foo')
3. }
```

vararg parameters are allowed and accessible as array values in the method body:

```
1. def printAll(String... strings) {
2.   strings.forEach[ s | println(s) ]
3. }
```

Recursive methods and abstract methods have to declare an explicit return type.

Abstract Methods

```
1. abstract class MyAbstractClass() {
2.   def String abstractMethod() // no body
3. }
```

Overriding Methods

If a method overrides a method from a super type, the override keyword is mandatory and replaces the keyword def.

```
1. override String second(List<String>
   elements) {
2.   elements.get(1)
3. }
```

Elvis Operator

Xtend supports the elvis operator known from Groovy.

```
1. val salutation = person.firstName ?:
   'Sir/Madam'
```

Null-Safe Feature Call

Checking for null references can make code very unreadable. Xtend supports the safe navigation operator ?. to make such code better readable.

```
1. if (myRef != null) myRef.doStuff()
```

is same as

```
1. myRef?.doStuff
```

Switch Expression

The switch expression is very different from Java's switch statement.

There is no fall through which means only one case is evaluated at most.

Second, switch can be used for any object reference. Object.equals(Object) is used.

```
1. switch myString {
2.    case myString.length > 5 : "long string."
3.    case 'some' : "It's some string."
4.    default : "It's another short string."
5. }
```

Type guards

The case only matches if the switch value conforms to this type. A case with both a type guard and a predicate only match if both conditions match. If the switch value is a field, parameter or variable, it is automatically casted to the given type within the predicate and the case body.

```
1. def length(Object x) {
2.    switch x {
3.      String case x.length > 0 : x.length
4.      // length is defined for String
5.      List<?> : x.size
6.      // size is defined for List
7.      default : -1
8.    }
9. }
```

@Property

Xtend compiler will generate a Java field, a getter and, if the field is non-final, a setter method. The name of the Java field will be prefixed with an _ and have the visibility of the Xtend field. The accessor methods are always public.

```
1. @Property String name
```

@Data

The annotation @Data (src), will turn an annotated class into a value object class. A class annotated with @Data is processed according to the following rules:

- all fields are final,
- getter methods will be generated (if they do not yet exist),
- a constructor with parameters for all non-initialized fields will be generated (if it does not exist),
- equals(Object) / hashCode() methods will be generated (if they do not exist),
- a toString() method will be generated (if it does not exist).

Property Access

If there is no field with the given name and also no method with the name and zero parameters accessible, a simple name binds to a corresponding Java-Bean getter method if available:

```
1. myObj.myProperty
2. // myObj.getMyProperty()
3. // (.. in case myObj.myProperty
4. // is not visible.)
```

Implicit Variables this and it

Like in Java the current instance of the class is bound to this. This allows for either qualified field access or method invocations like in:

```
1. this.myField
```

or it is possible to omit the receiver:

```
1. myField
```

You can use the variable name it to get the same behavior for any variable or parameter:

```
1. val it = new Person
2. name = 'Horst'
3. // translates to 'it.setName("Horst");'
```

Another speciality of the variable it is that it is allowed to be shadowed. This is especially useful when used together with lambda expressions.

As this is bound to the surrounding object in Java, it can be used in finer-grained constructs such as lambda expressions. That is why it.myProperty has higher precedence than this.myProperty.

Static Access

For accessing a static field or method you have to use the double colon :: like in this example:

```
1. MyClass::myField
2. com::acme::MyClass::myMethod('foo')
```

Alternatively you could import the method using a static import.

Template Expressions

Templates allow for readable string concatenation. Templates are surrounded by triple single quotes (''). A template expression can span multiple lines and expressions can be nested which are evaluated and their toString() representation is automatically inserted at that position.

The terminals for interpolated expression are so called guillemets «expression». They read nicely and are not often used in text so you seldom need to escape them. These escaping conflicts are the reason why template languages often use longer character sequences like e.g. <%= expression %> in JSP, for the price of worse readability. The downside with the guillemets in Xtend is that you will have to have a consistent encoding. Always use UTF-8 and you are good.

If you use the Eclipse plug-in the guillemets will be inserted on content assist within a template. They are additionally bound to CTRL+SHIFT+< and CTRL+SHIFT+ for « and » respectively. On a Mac they are also available with alt+q («) and alt+Q (»).

Let us have a look at an example of how a typical method with a template expressions looks like:

```
1. def someHTML(String content) '''
2.   <html>
3.     <body>
4.       «content»
5.     </body>
6.   </html>
7. '''
```

As you can see, template expressions can be used as the body of a method. If an interpolation expression evaluates to null an empty string is added.

Template expressions can occur everywhere. Here is an example showing it in conjunction with the powerful switch expression:

```
1. def toText(Node n) {
2.   switch n {
3.     Contents : n.text
4.
5.     A : '''<a
   href="«n.href»">«n.applyContents»</a>'''
6.
7.     default : '''
8.         <«n.tagName»>
9.           «n.applyContents»
10.        </«n.tagName»>
11.    '''
12.  }
13. }
```

Conditions in Templates

There is a special IF to be used within templates:

```
1. def someHTML(Paragraph p) '''
2.   <html>
3.     <body>
4.       «IF p.headLine != null»
5.         <h1>«p.headline»</h1>
6.       «ENDIF»
7.       <p>
8.         «p.text»
9.       </p>
10.    </body>
11.  </html>
12. '''
```

Loops in Templates

Also a FOR expression is available:

```
1. def someHTML(List<Paragraph> paragraphs)
   '''
2.   <html>
3.     <body>
4.       «FOR p : paragraphs»
5.         «IF p.headLine != null»
6.           <h1>«p.headline»</h1>
7.         «ENDIF»
8.         <p>
9.           «p.text»
10.        </p>
11.      «ENDFOR»
12.    </body>
13.  </html>
14. '''
```

The for expression optionally allows to specify what to prepend (BEFORE), put in-between (SEPARATOR), and what to put at the end (AFTER) of all iterations. BEFORE and AFTER are only executed if there is at least one iteration. (SEPARATOR) is only added between iterations. It is executed if there are at least two iterations.

```
1. def someHTML(List<Paragraph> paragraphs)
   '''
2.   <html>
3.     <body>
4.       «FOR p : paragraphs BEFORE '<div>'
   SEPARATOR '</div><div>' AFTER '</div>'»
5.         «IF p.headLine != null»
6.           <h1>«p.headline»</h1>
7.         «ENDIF»
8.         <p>
9.           «p.text»
10.        </p>
11.      «ENDFOR»
12.    </body>
13.  </html>
14. '''
```

Typing

The template expression is of type CharSequence. It is automatically converted to String if that is the expected target type.

White Space Handling

One of the key features of templates is the smart handling of white space in the template output. The white space is not written into the output data structure as is but preprocessed. This allows for readable templates as well as nicely formatted output. The following three rules are applied when the template is evaluated:

1. Indentation in the template that is relative to a control structure will not be propagated to the output string. A control structure is a FOR-loop or a condition (IF) as well as the opening and closing marks of the template string itself. The indentation is considered to be relative to such a control structure if the previous line ends with a control structure followed by optional white space. The amount of indentation white space is not taken into account but the delta to the other lines.
2. Lines that do not contain any static text which is not white space but do contain control structures or invocations of other templates which evaluate to an empty string, will not appear in the output.
3. Any newlines in appended strings (no matter they are created with template expressions or not) will be prepended with the current indentation when inserted.

Although this algorithm sounds a bit complicated at first it behaves very intuitively. In addition the syntax coloring in Eclipse communicates this behavior.

```
def someHTML(List<Paragraph> paragraphs) '''
  <html>
    <body>
      «FOR p : paragraphs BEFORE '<div>' SEPARATOR '</div><div' AFTER '</div>'»
        «IF p.headline != null»
          <h1>«p.headline»</h1>
        «ENDIF»
        <p>
          «p.text»
        </p>
      «ENDFOR»
    </body>
  </html>
'''
```

The behavior is best described with a set of examples. The following table assumes a data structure of nested nodes.

```
1. class Template {
2.   def print(Node n) '''
3.     node «n.name» {}
4.   '''
5. }
```

```
1. node NodeName {}
```

The indentation before node «n.name» will be skipped as it is relative to the opening mark of the template string and thereby not considered to be relevant for the output but only for the readability of the template itself.

```
1. class Template {
2.   def print(Node n) '''
3.     node «n.name» {
4.       «IF hasChildren»
5.         «n.children.map[print]»
6.       «ENDIF»
7.     }
8.   '''
9. }
```

```
1. node Parent{
2.   node FirstChild {
3.   }
4.   node SecondChild {
5.     node Leaf {
6.     }
7.   }
8. }
```

As in the previous example, there is no indentation on the root level for the same reason. The first nesting level has only one indentation level in the output. This is derived from the indentation of the IF hasChildren condition in the template which is nested in the node. The additional nesting of the recursive invocation children.map[print] is not visible in the output as it is relative the the surrounding control structure. The line with IF and ENDIF contain only control structures thus they are skipped in the output. Note the additional indentation of the node Leaf which happens due to the first rule: Indentation is propagated to called templates.