Classes, fields, methods

several classes can be defined in a file

```
class ExampleClass {
    def static void main(String[] args) {
        println(”Hello, World!”)
    }
}
```

```
class ExampleClass {
    fields are private by default
    var x = 0;
    var y = 0; optional semicolons after statements
    methods defined either with def or override
    def int sum() {
        return x + y
    }

    return type can be omitted if it can be inferred
    return type should be declared for recursive and abstract methods
    def mul() {
        x * y  last expression in method's body is the return expression
    }

    def max() {
        if (x>=y) x else y
    }
    constructor
    new () {}
        types of method parameters must always be specified
    new (int aX, int aY) {
        x = aX
        y = aY
    }

    override String toString() {
        ’’’(«x», «y»)’’’
    }       template expression with two placeholders for values of x and y
}
```

```
def square(int x) {
    final variable
    val text = “The square is: ”
    non-final variable
    var s = x * x
    assignment to a final variable not allowed
    text = “New text: ”
    non-final variables can be assigned
    s = s + 0

    text + s
}
```

```
variable number of arguments (varargs)
those arguments are accessible as array values
def printAll(String... strings) {
    strings.forEach[println(it)]
}                    lambda expression
```

```
                    empty parentheses are optional
new String() == new String
```

Implicit variable `it`

```
implicit variable it
val it = new Person
name = “John”
```
corresponds to `it.setName(”John”)` in Java

```
class ItExample {          implicit variable it
    def doSomething(String it) {
        toLowerCase
    }
}
```
corresponds to `return it.toLowerCase()` in Java

Getters and setters for fields

```
println(o.name)      o.getName()
          getter

o.name = “John”      o.setName(”John”)
setter
```

switch expressions

```
def String switchExample(Entity e, Entity specialEntity) {
    switch value: any object reference
    switch e { only the selected case is executed
                in Java: switch falls through all matching cases
        Boolean expression: case matches if expression evaluates to true
        case e.name.length > 0 : “has a name”
                                 no break statements
        case e.superType != null : “has super type”

        case e.name.startsWith(”_”), multiple cases fall-through
        case e.name.startsWith(”#”):
             “starts with a special symbol”
        for non-Boolean expressions: compare using equals
        case specialEntity : “equals to special entity”

        default : “nothing”
    } cases are evaluated in the specification order
}
```

switch expressions as type guards

```
class Animal {}
class Cat extends Animal {}
class Dog extends Animal {}

def toString(Animal x) {
    type guard
    switch x { case matches only if switch value conforms to type
               specified in the case
        is x an instance of Cat?
        Cat : “cat”
        is x an instance of Dog?
        Dog : “dog”
    }
}
```

Polymorphic method invocation

```
class AnyType {}
class NumberType extends AnyType {}
class TextType extends AnyType {}

class Example {

    without dispatch: method overloading
                      selection of the specific method according to static types of arguments
    multiple dispatch: method is selected according to runtime type of arguments
    def dispatch typeToString(NumberType t) { “integer” }
                 note: typeToString is not a method of either of AnyType, NumberType or TextType
    def dispatch typeToString(TextType t) { “string” }

    def run() {
                 static type of a is AnyType    runtime type of a is NumberType
        var AnyType a = new NumberType
        println(”The type is ” + typeToString(a))
    }                                invoked on an instance of AnyType:
}                                    actual type of that instance is NumberType
                                     thus “integer” will be returned
```

Extension methods

```
MyType x

f is not declared in MyType
f(x)

    extension method
    f is invoked as if it were a method of class MyType
x.f()
    Uniform Function Call Syntax (UFCS)

    empty parentheses can be omitted
x.f
```

```
class MyExampleClass {
    extension field of type MyMathUtils in class MyExampleClass
    methods of MyMathUtils become extension methods in MyExampleClass
    extension MyMathUtils f
                          field name can be omitted
    val x = 0
    val String s = “John”
    var Boolean b = false

    def method1() {
        x.increase
    }     defined in MyMathUtils for instances of integer

    def method2() {
        extension variable
        methods of MyStringUtils become extension methods in current code block
        var extension MyStringUtils z
        s.appendHello
    }     defined in MyStringUtils for instances of String

                  extension parameter
                  methods of MyBoolUtils become extension methods in the method
    def method3(extension MyBoolUtils z) {
        b.negate.negate.negate
    }     defined in MyBoolUtils for instances of Boolean
}
```

```
class MyMathUtils {
    def increase(int x) { x+1 }
}

class MyBoolUtils {
    def negate(Boolean b) { !b }
}

class MyStringUtils {
    def appendHello(String s) { “Hi ” + s }
}
```

```
class MyMathUtils2 {
    methods defined in an Xtend class can be automatically used as extension methods with the class
    def increase(int x) { x+1 }
    def increaseTwoTimes(int x) { x.increase.increase }
}
```

Lambda expressions

```
                lambda expression  can be stored in a variable
val l = [String s, int i | s + i]
          parameters       body

println(l.apply(”s”, 10))
          evaluates a lambda expression
```

```
    function type
val (String, int)=>String l = [String s, int i | s + i]
    types of parameters  return type
                          types of parameters can be omitted
                          they are redundant and can be inferred
                          because the type of lambda expression is explicitly specified
```

```
                methods takes a lambda expression as a parameter; its type is (String, int)↦String
def execute( (String, int)=>String f ) {
    f.apply(”s”, 10)
}
execute([s, i | s + i])
          types of parameters are redundant since they can be inferred
```

```
                                    compliant with int compare(T obj1, T obj2)
Collections.sort( list,             [arg0, arg1 | arg0.CompareToIgnoreCase(arg1)] )
expects a List<T> and a Comparator<T>   lambda expression compliant with the single abstract method of a functional interface
Comparator<T> is a functional interface
functional interface: SAM (Single Abstract Method) type
it has single abstract method int compare(T obj1, T obj2)
```

```
Collections.sort( list ) [arg0, arg1 | arg0.CompareToIgnoreCase(arg1)]
    when lambda expression is the last argument in method invocation, it can be put outside the parentheses
    the parentheses can be omitted at all if the invocation only requires one argument

Collections.sort( list ) [$0.CompareToIgnoreCase($1)]
    if parameters of a lambda expression can be inferred from the context, their names can be omitted
    in that case, parameters can be referred to by using $0, $1, etc.
```

```
mystrings.findFirst[s | s.startsWith(”T”)]
    if a lambda expression has only one parameter, its declaration can be omitted:
    in that case, it can be used an the implicit parameter

mystrings.findFirst[it.startsWith(”T”)]
    it is the default parameter name in a lambda expression

mystrings.findFirst[startsWith(”T”)]
    all members of it are implicitly available, thus it can be omitted
```

with operator

corresponding Java code:
```
val person = eINSTANCE.createPerson
person.name = “John”
person.surname = “Smith”
return person
```

```
return eINSTANCE.createPerson =>
    [                                with operator
        name = “John”                binds an object to the scope
        surname = “Smith”            of a lambda expression
    ]
```

```
=>
with operator is a binary operator
left operand: expression                    it can be omitted
right operand: lambda expression with single parameter
applies lambda to the left operand
result: left operand after applying lambda
```

Cheat sheet on Eclipse Xtend

by Mikhail Barash
http://dsl-course.org

Based on L. Bettini's book *Implementing domain-specific languages with Xtext and Xtend*