DSLs for Java Developers

Installation Instructions

1. Download the latest and greatest Eclipse version or simply use the distribution of your choice. Anything newer than Eclipse 3.5 will do.
2. Add the Xtext milestone update site preferably the one from itemis as it contains the Antlr generator: http://download.itemis.com/updates/milestones/
3. Select the Xtext SDK and hit *Install*.

Tutorial Outline

1. Use the scaffolded tutorial language to get used to the editor.
2. Allow to use Java types in the DSL.
3. Get familiar with the Java types model and Xtend.

Coffee Break

4. Make Java types and entities usable interchangeably.
5. Introduce operations with expression bodies.

Where do we want to go today?

In this tutorial you will learn how to create a domain-specific language that tightly integrates with the Java development tools, allows to use Java classes together with the domain-specific abstractions, supports behavioral expressions, and compiles to executable Java code. Concepts such as the JVM model inference and the scoping of expressions will be explained along with the big picture of the Xtext framework.

```
import java.util.*

package tutorial {
  entity Person {
    firstName: String
    lastName: String
    friends: List<Person>
    birthday: Date
    op getFullName(): String {
       firstName + " " + lastName
    }
    op getSortedFriends(): List<Person> {
       friends.sortBy[ p | p.fullName ]
    }
  }
}
```


Exercise 1: Scaffold the Tutorial Language

1. Select the *EclipseCon 2012 - Xtext Tutorial* from the *New Example* wizard.
    1. Choose *File > New ...*
    2. Navigate to *Example...*
    3. Select the *Xtext Examples > EclipseCon 2012 - Xtext Tutorial* and *Finish* the wizard.
2. Find the file *Tutorial.xtext* in the newly created projects.
3. Choose Run > Generate Artifacts from the context menu.
4. Try the editor of the tutorial language.
    1. Go to the context menu of one of the projects.
    2. Select *Run > RunConfigurations* and run *Eclipse Application > Launch Runtime Eclipse.*
    3. Create a new Java project *sample* in the spawned Eclipse instance.
    4. Select the *src*-folder in that project and create a new file *sample.tutorial*. **It's important to use the file extension 'tutorial'.** When asked for adding the Xtext nature select *Yes*.
    5. Try the editor features and create some example content.

```
import types.*

package tutorial {
  entity Person {
    firstName: String
    lastName: String
  }
}

package types {
  type String
}
```

You may want to study the grammar file *Tutorial.xtext* to become familiar with the syntax and semantics of a language definition. Things you may need in the next exercises:

- *Alternatives:* The bar operator | is used to describe alternatives in a production rule. Note that the following two examples have exactly the same semantics:

```
a b | c d
(a b) | (c d)
```

- *Cardinalities:* The question mark is used as a cardinality operator to denote zero or one occurrences. The star means zero or many while the plus operator means one or more. If there is no cardinality specified exactly one occurrence is valid:

```
a?        a*        a+        a
```

- *Assignments:* Assignments can be used to populate the objects that the parser produces in memory. Important distinctions are the multi value assignment and the single value assignment. While the former adds values to a list of things, the latter will assign a single value by means of a setter:

```
properties+=Property        name=ID
```

Exercise 2: Replace simple data types by real Java classes and allow entities to extend Java types

For now the tutorial languages feels sort of alien in a Java project. Even the simplest things such as a the types *String* or *Date* have to be redefined. We want to use existing Java types instead.

```
package tutorial {
  import java.util.Date
  entity Person {
    firstName: String
    lastName: String
    birthday: Date
  }
}
```

1. Remove all occurrences of the parser rules *DataType* and *Type*.
2. Replace the cross references to *Entity* and *Type* with invocations of the inherited production rule *JvmTypeReference*. This rule encapsulates the logic to parse and create type references as you are used from Java. All the fluff with generics, primitives and arrays is supported.
3. Regenerate the language and try to update your model files.

✳ You will notice that the Java development tools of Eclipse are aware of the type references that you use in your tutorial files. It is possible to refer to own classes, too. You may want to try to create an enum type *Gender* to define the sex of a person. Try *Find references* on the *Gender* class afterwards.

```
package tutorial {
  import java.util.Date
  import tutorial.types.*
  entity Person {
    firstName: String
    lastName: String
    birthday: Date
    gender: Gender
  }
}
```

```
package tutorial.types;

public enum Gender {
   MALE, FEMALE
}
```

Exercise 3: Learn how to instantiate artificial Java types

Part of the Xtext framework is the *JvmTypesBuilder*, a powerful API to create Java types on the fly. The idea is to derive a number of JVM elements from parsed source elements.

1. The *JvmTypesBuilder* can be used to construct the Java class that you would usually write manually to implement a concept such as *Person*. Take a look at the generated Xtend file *TutorialJvmModelInferrer*. Notice how the *JvmTypesBuilder* is used to create a JVM model.
2. The *TypesBuilderExercise* in the test plug-in is a JUnit 4 test also written in Xtend. It checks if your JVM model is complete by comparing the inferred Java code to the expected Java code below. Note how easy that is using Xtend's multiline string literals. You can run it with *Run As > JUnit Test*.
3. Your task is to implement the creation of a JVM model conforming to the Java code below, ignoring the *sourceElement* for now. The *TypesBuilderExercise* will help you to get the *TutorialJvmModelInferrer* right step by step.

✶ It is possible to associate arbitrary expression code with the produced structures. Try to play around with support for property change listeners or invariants, e.g. fields may not become null.

```
package tutorial;

import java.util.Date;

/**
 * A simple entity to describe a Person
 */
public class Person {
  private String firstName;

  public String getFirstName() {
    return this.firstName;
  }

  public void setFirstName(final String firstName) {
    this.firstName = firstName;
  }
...
  private Date birthday;

  public Date getBirthday() {
    return this.birthday;
  }

  public void setBirthday(final Date birthday) {
    this.birthday = birthday;
  }
}
```

Exercise 4: Translate entities to Java

Now that you are familiar with the *JvmTypesBuilder*, it is time to translate the actual entities to Java classes automatically. This will save a lot of typing and opens the door to a tight integration of the DSL with the Java development tools.

1. Specialize the method *infer* in the *TutorialJvmModelInferrer* to take an *Entity* instead of an instance of *DomainModelTutorial*. The multiple dispatch logic of Xtend will invoke the method with the right arguments.
2. Transform the given *Entity* to a *JvmGenericType* and its properties to fields, getters and setters. To calculate the qualified name of an *Entity* you can use the *IQualifiedNameProvider*. Mind documentation comments that should not get lost.
3. Restart Eclipse and try the new features, such as using entities as type arguments for lists.

```
package tutorial {
  import java.util.*
  import tutorial.types.*

  entity Person {
    firstName: String
    lastName: String
    friends: List<Person>

    birthday: Date
    gender: Gender
  }
}
```

✶ If you are keen on hacking more Xtend code, try to synthesize additional elements in the Java class such as

- a constructor that takes arguments,
- an equals or a hashCode routine or
- annotations such as JDTs new *@NonNull* or *@Nullable* stuff.

✶✶ Think about more options that should be available for entities. Do you want to define abstract entities? Mandatory fields could be useful, too? Go ahead and update the grammar definition along with the model inferrer.

Exercise 5: Putting it all together Allow to use expressions in entities

To make the language really powerful, it's time to facilitate expressions and allow to define operations for entities. An operation defines a number of formal parameters, a return type and a body.

1. Add *Operation* as an alternative to the *Property* concept of the tutorial language.
   Note that the Xbase super grammar already provides a definition of the concrete syntax for *FullJvmParameter* and *XBlockExpression*.

2. Don't forget to regenerate the language infrastructure.

3. Update the model inferrer to create Java methods from your operation concept. Also transfer the body of the operation to the body of the newly produced method. This assigns the logical container to the block and allows to resolve the parameters and type arguments and to apply visibility rules.

4. The expressions inside the operation bodies rely on a runtime library. The project *sample* in your runtime Eclipse needs to have that library on its classpath. To accomplish this, right click on the project, go to *Properties* > *Java Build Path* > *Libraries*, and choose *Add Library* > *Xtend Library*.

5. Try the editor and see how Java and your own language interact with each other.

```
import java.util.*

package tutorial {
  entity Person {
    firstName: String
    lastName: String
    friends: List<Person>
    birthday: Date
    op getFullName(): String {
       firstName + " " + lastName
    }
    op getSortedFriends(): List<Person> {
       friends.sortBy[ p | p.fullName ]
    }
  }
}
```