DSLs for Java Developers

Tutorial Outline

1. Become familiar with the grammar language.
2. Run the code generator and try the editor.
3. Translate your DSL to Java.

Coffee Break

4. Add support for operations.
5. More Q & A.

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.

```
package tutorial

import java.util.List

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

Exercise 1: Scaffold the Tutorial Language

1. Import the *Xtext Tutorial* from USB key.
    1. Choose *File > Import …*
    2. Navigate to *General > Existing Projects into Workspace*
    3. Choose *Select archive file:* and browse to the provided *zip Archive*
    4. Select the *Xtext_Tutorial.zip* and hit Finish.
2. Find the file *Tutorial.xtext* in the imported 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.

```
package tutorial

import java.util.Date

entity Person {
    firstName: String
    lastName: String
    birthday: Date
}
```

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: Translate your language to Java

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. This will save a lot of typing and opens the door to a tight integration of the DSL with the Java development tools.

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. Specialize the method infer in the *TutorialJvmModelInferrer* to take an *Entity* instead of an instance of *DomainModelFile*. The multiple dispatch logic of Xtend will invoke the method with the right arguments.

4. Your task is to implement the creation of a JVM model conforming to the expected Java code. The *TypesBuilderExercise* will help you to get the *TutorialJvmModelInferrer* right step by step. Therefore you have to 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 that documentation comments should not get lost.

✳ 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.

✳✳ 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.

```
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;
  }
...
}
```

Exercise 3: 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.

```
package tutorial

import java.util.List

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