Implementing Domain-Specific Languages with Xtext and Xtend

Learn how to implement a DSL with Xtext and Xtend
using easy-to-understand examples and best practices

Lorenzo Bettini

Implementing Domain-Specific Languages with Xtext and Xtend

Learn how to implement a DSL with Xtext and Xtend using easy-to-understand examples and best practices

Lorenzo Bettini

PACKT PUBLISHING

open source*
community experience distilled

Implementing Domain-Specific Languages with Xtext and Xtend

Credits

Author
Lorenzo Bettini

Reviewers
Dr. Jan Koehnlein
Henrik Lindberg
Pedro J. Molina
Sebastian Zarnekow

Acquisition Editor
James Jones

Lead Technical Editor
Neeshma Ramakrishnan

Technical Editors
Dipika Gaonkar
Anita Nayak
Sonali S. Vernekar

Project Coordinators
Shiksha Chaturvedi
Hardik Patel

Proofreader
Paul Hindle

Indexer
Rekha Nair

Graphics
Sheetal Aute
Ronak Dhruv
Abhinash Sahu

Production Coordinator
Aparna Bhagat
Kirtee Shingan

Cover Work
Aparna Bhagat

About the Author

Lorenzo Bettini is an assistant professor (Researcher) in computer science
at Dipartimento di Informatica, Università di Torino, Italy. Previously, he was
a Postdoc and a contractual researcher at Dipartimento di Sistemi e Informatica,
Università di Firenze, Italy.

He has a Masters Degree in computer science and a PhD in theoretical computer science.

His research interests cover design, theory, and the implementation of programming
languages (in particular, object-oriented languages and network-aware languages).

He has been using Xtext since version 0.7. He has used Xtext and Xtend
for implementing many Domain Specific Languages and Java-like
programming languages.

He is also the author of about 60 papers published in international conferences
and international journals.

You can contact him at `http://www.lorenzobettini.it`.

Acknowledgement

About the Reviewers

Dr. Jan Koehnlein is a core committer of the Xtext project and the Xtend language. He has several years of experience in model-driven software development and on the Eclipse platform. Jan is currently working as a consultant and software architect for itemis in Germany.

Henrik Lindberg has worked for many companies since the early 80s, and he has had the opportunity to work with most aspects of software development from operating systems to applications as a developer, architect, CTO, and founder.

He is currently CTO and founder of Cloudsmith Inc, a Puppet Labs Inc. partner specializing in services and tools for the creation, testing, and deployment of software stacks in local and cloud infrastructures. Prior to Cloudsmith, he ran the BEA/JRockit development office (now Oracle's JVM division).

Henrik always had a passion for computer languages and parser technology, and he has worked with Eclipse Xtext on several projects, most recently the Puppet Language IDE called Geppetto. He is an Eclipse committer on Eclipse p2 and leads the Eclipse Buckminster and b3 projects. He is a frequent contributor to the Xtext forum.

You can contact him on Twitter as `@hel` and also on the Eclipse and Puppet IRCs with the tag `helindbe` as well as on the Eclipse forums.

> I would like to thank Carl Barks and The Junior Woodchucks Guidebook that in an early Swedish edition contained a section with the Woodchucks Crypto—this inspired me to invent several written and spoken languages that I tried to teach my friends. This had limited success, we were after all only 7 years old.

Pedro J. Molina is a practitioner and researcher in the field of model-driven development. From his masters thesis in 1998 to his PhD in 2004, he worked on the research of conceptual user interface patterns for code generation on business applications and published more than 20 papers and 2 books.

Within industry, he has been working for CARE Technologies developing commercial code generators and doing consultancy and software architecture for Capgemini. Nowadays, he is the chief architect officer for Icinetic, a firm building modeling and code generation tools with a strong focus on architecture.

Pedro has been taking part in the program committee for Code Generation Conference for the last 7 years and keeps up-to-date with industrial MDD efforts. He maintains a blog: The Metalevel, where he talks about MDD and code-generation at `http://pjmolina.com/metalevel`.

www.PacktPub.com

Support files, eBooks, discount offers, and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?
- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

Xtext is an open source Eclipse framework for implementing Domain Specific Languages together with their integration in the Eclipse IDE. Xtext lets you implement languages quickly by covering all aspects of a complete language infrastructure, starting from the parser, code generator, or interpreter, up to a full Eclipse IDE integration (with all the typical IDE features such as editor with syntax highlighting, code completion, error markers, automatic build infrastructure, and so on).

This book will incrementally guide you through the very basics of DSL implementation with Xtext and Xtend, such as grammar definition, validation, and code generation; the book will then cover advanced concepts such as unit testing, type checking, and scoping. Xtext comes with good and smart default implementations for all these aspects. However, every single aspect can be customized by the programmer.

Although Java can be used for customizing the implementation of a DSL, Xtext fosters the use of Xtend, a Java-like programming language completely interoperable with the Java type system which features a more compact and easier to use syntax and advanced features such as type inference and lambda expressions. For this reason, we will use Xtend throughout the book.

Most of the chapters have a tutorial nature and will describe the main concepts of Xtext through uncomplicated examples. The book also uses test driven development extensively.

This book aims at being complementary to the official documentation, trying to give you enough information to start being productive in implementing a DSL with Xtext. This book will try to teach you some methodologies and best practices when using Xtext, filling some bits of information that are not present in the official documentation.

The chapters are meant to be read in order, since they typically refer to concepts that were introduced in the previous chapters.

All the examples shown in the book are available online, see the section *Downloading the example code*. We strongly suggest that you first try to develop the examples while reading the chapters and then compare their implementations with the ones provided by the author.

What this book covers

After a small introduction to the features that a DSL implementation should cover (including integration in an IDE), the book will introduce Xtend since it will be used in all the examples. The book proceeds by explaining the main concepts of Xtext; for example, validation, code generation, and customizations of runtime and UI aspects. The book will then show how to test a DSL implemented in Xtext with Junit in order to follow a Test Driven Development strategy that will help you to quickly implement cleaner and more maintainable code. The test-driven approach is used in the rest of the book when presenting advanced concepts such as type checking and Scoping. The book also shows how to build and release a DSL so that it can be installed in Eclipse and hints on how to build the DSL headlessly in a continuous integration server. The last chapter briefly introduces Xbase.

Chapter 1, *Implementing a DSL,* gives a brief introduction to Domain Specific Languages (DSL) and sketches the main tasks for implementing a DSL and its integration in an IDE. The chapter also shows how to install Xtext and gives a first idea of what you can do with Xtext.

Chapter 2, *Creating Your First Xtext Language*, shows a first example of a DSL implemented with Xtext and gives an introduction to some features of the Xtext grammar language. The chapter describes the typical development workflow of programming with Xtext and provides a small introduction to EMF (Eclipse Modeling Framework), a framework on which Xtext is based.

Chapter 3, *The Xtend Programming Language*, describes the main features of the Xtend programming language, a Java-like language interoperable with Java. We will use Xtend to implement every aspect of an Xtext DSL.

Chapter 4, *Validation*, describes validation, in particular, the Xtext mechanism to implement validation, that is, the validator. This chapter is about implementing additional constraint checks that cannot be done at parsing time. It also shows how to implement quickfixes corresponding to the errors generated by the validator.

Chapter 5, *Code Generation*, shows how to write a code generator for an Xtext DSL using the Xtend programming language. The chapter also shows how a DSL implementation can be exported as a Java standalone command-line compiler.

Chapter 6, *Customizations*, describes the main mechanism for customizing Xtext components, Google Guice, a Dependency Injection framework. In particular, the chapter shows how to customize both the runtime and the UI aspects of an Xtext DSL.

Chapter 7, *Testing*, describes how to test a DSL implementation using Junit and the additional utility classes provided by Xtext. The chapter shows the typical techniques for testing both the runtime and the UI aspects of a DSL implemented in Xtext.

Chapter 8, *An Expression Language*, covers the implementation of a DSL for expressions, including arithmetic, boolean, and string expressions. The chapter shows how to deal with recursive rules and with typical problems when writing Xtext grammars. The implementation will be described incrementally and in a test-driven way. The chapter also shows how to implement a type system for checking that expressions are correct with respect to types and how to implement an interpreter for these expressions.

Chapter 9, *Type Checking*, covers the implementation of a small object-oriented DSL, which can be seen as a smaller version of Java that we call SmallJava. This chapter shows how to implement some type checking techniques that deal with object-oriented features, such as inheritance and subtyping.

Chapter 10, *Scoping*, covers the main mechanism behind visibility and cross-reference resolution in Xtext. Since scoping and typing are often strictly connected and inter-dependent especially for object-oriented languages, the chapter is based on the SmallJava DSL introduced in the previous chapter. The chapter describes both local and global scoping and how to customize them.

Chapter 11, *Building and Releasing*, describes how you can release your DSL implementation by creating an Eclipse p2 repository so that others can easily install it in Eclipse. The chapter is based on the Xtext wizard that creates the infrastructure to build a p2 repository with Buckminster. The wizard will also create all the needed files to build your projects and test them in a headless way so that you can easily run your builds in a continuous integration server.

Chapter 12, *Xbase*, describes Xbase a reusable expression language interoperable with Java. By using Xbase in your DSL, you will inherit all the Xbase mechanisms for performing type checking according to the Java type system and the automatic Java code generation.

What you need for this book

The book assumes that you have a good knowledge of Java; it also assumes that you are familiar with Eclipse and its main features. Existing basic knowledge of a compiler implementation would be useful, though not strictly required, since the book will explain all the stages of the development of a DSL.

Who this book is for

This book is for programmers who want to learn about Xtext and how to use it to implement a DSL or a programming language together with the Eclipse IDE tooling.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "We can include other contexts through the use of the `include` directive".

A block of code is set as follows:

```
public static void main(String args[]) {
    System.out.println("Hello world");
```

Where **keywords** of the languages are typeset in bold and static members are typeset in italics (for example, Java static methods and fields).

Bibliographic references are of the form "Author" "year" when there is a single author, or "First author et al." "year" when there is more than one author. Bibliographic references are used for books, printed articles, or articles published on the Web. The Bibliography can be found at the end of the book.

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "clicking on the **Next** button moves you to the next screen". When the user is requested to select submenus, we separate each menu with a pipe, like this: "To create a new project, navigate to **File** | **New** | **Project...**".

> Warnings or important notes appear in a box like this.

> Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message. If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

The example code for this book is also available on a Git repository at `https://github.com/LorenzoBettini/packtpub-xtext-book-examples`.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Implementing a DSL

In this chapter, we will give a brief introduction on Domain Specific Languages (DSL) and the issues concerning their implementation, especially in the context of an IDE. The initial part of the chapter is informal: we will sketch the main tasks for implementing a DSL and its integration in an IDE. At the end of the chapter, we will also show you how to install Xtext and will give you a glimpse of what you can do with Xtext. Xtext is an Eclipse framework for the development of DSLs that covers all aspects of a language implementation, including its integration in the Eclipse IDE.

Domain Specific Languages

Domain Specific Languages, abbreviated as **DSL**, are programming languages or specification languages that target a specific problem domain. They are not meant to provide features for solving all kinds of problems; you probably will not be able to implement all programs you can implement with, for instance, Java or C (which are known as **General Purpose Languages**). But if your problem's domain is covered by a particular DSL, you will be able to solve that problem easier and faster by using that DSL instead of a general purpose language.

Some examples of DSLs are SQL (for querying relational databases), Mathematica (for symbolic mathematics), HTML, and many others you have probably used in the past. A program or specification written in a DSL can then be interpreted or compiled into a general purpose language; in other cases, the specification can represent simply data that will be processed by other systems.

For a wider introduction to DSLs, you should refer to Fowler 2010, Ghosh 2010, and Voelter 2013.

So, why should you create a new language?

You may now wonder why you need to introduce a new DSL for describing specific data, for example, models or applications, instead of using XML, which allows you to describe data in a machine in human-readable form. There are so many tools now that, starting from an XML schema definition, allow you to read, write, or exchange data in XML without having to parse such data according to a specific syntax. There is basically only one syntax to learn (the XML tag syntax) and then all data can be represented with XML.

Of course, this is also a matter of taste, but many people (including the author himself) find that XML is surely machine-readable, but not so much human-readable. It is fine to exchange data in XML if the data in that format is produced by a program. But often, people (programmers and users) are requested to specify data in XML manually; for instance, for specifying an application's specific configuration.

If writing an XML file can be a pain, reading it back can be even worse. In fact, XML tends to be verbose, and it fills documents with too much additional syntax noise due to all the tags. The tags help a computer to process XML, but they surely distract people when they have to read and write XML files.

Consider a very simple example of an XML file describing people:

```
<people>
  <person>
    <name>James</name>
    <surname>Smith</surname>
    <age>50</age>
  </person>
  <person employed="true">
    <name>John</name>
    <surname>Anderson</surname>
    <age>40</age>
  </person>
</people>
```

It is not straightforward for a human to grasp the actual information about a person from such a specification: a human is distracted by all those tags. Also, writing such a specification may be a burden. An editor might help with some syntax highlighting and early user feedback concerning validation, but still there are too many additional details.

How about this version written in an ad-hoc DSL?:

```
person {
  name=James
  surname=Smith
  age=50
}
person employed {
  name=John
  surname=Anderson
  age=40
}
```

This contains less noise and the information is easier to grasp. We could even do better and have a more compact specification:

```
James Smith (50)
John Anderson (40) employed
```

After all, since this DSL only lets users describe the name and age of people, why not design it to make the description both compact and easy to read?

Implementing a DSL

For the end user, using a DSL is surely easier than writing XML code; however, the developer of the DSL is now left with the task of implementing it.

Implementing a DSL means developing a program that is able to read text written in that DSL, parse it, process it, and then possibly interpret it or generate code in another language. Depending on the aim of the DSL, this may require several phases, but most of these phases are typical of all implementations.

In this section we only hint at the main concepts of implementing a DSL.
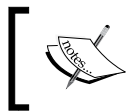
> From now on, throughout the book we will not distinguish, unless strictly required by the context, between *DSL* and *programming language*.

Parsing

First of all, when reading a program written in a programming language, the implementation has to make sure that the program respects the syntax of that language.

To this aim, we need to break the program into tokens. Each token is a single atomic element of the language; this can be a **keyword** (such as `class` in Java), an **identifier** (such as a Java class name), or **symbol name** (such as a variable name in Java).

For instance, in the preceding example, `employed` is a keyword; the parentheses are operators (which can be seen as special kinds of keywords as well). All the other elements are **literals** (`James` is a string literal and `50` is an integer literal).

The process of converting a sequence of characters into a sequence of tokens is called **lexical analysis**, and the program or procedure that performs such analysis is called a **lexical analyzer**, **lexer**, or simply a **scanner**. This analysis is usually implemented by using regular expressions syntax.

Having the sequence of tokens from the input file is not enough: we must make sure that they form a valid statement in our language, that is, they respect the syntactic structure expected by the language.
This phase is called **parsing** or **syntactic analysis**.

Let us recall the DSL to describe the name and age of various people and a possible input text:

```
James Smith (50)
John Anderson (40) employed
```

In this example each line of the input must respect the following structure:

- two string literals
- the operator `(`
- one integer literal
- the operator `)`
- the optional keyword `employed`

In our language, tokens are separated by white spaces, and lines are separated by a newline character.

You can now deduce that the parser relies on the lexer.

If you have never implemented a programming language, you might be scared at this point by the task of implementing a parser, for instance, in Java. You are probably right, since it is not an easy task. The DSL we just used as an example is very small (and still it would require some effort to implement), but if we think about a DSL that has to deal also with, say, arithmetic expressions? In spite of their apparently simple structure, arithmetic expressions are recursive by their own nature, thus a parser implemented in Java would have to deal with recursion as well (and, in particular, it should avoid endless loops).

There are tools to deal with parsing so that you do not have to implement a parser by hand. In particular, there are DSLs to specify the grammar of the language, and from this specification they automatically generate the code for the lexer and parser (for this reason, these tools are called **parser generators** or **compiler-compilers**). In this context, such specifications are called grammars. A **grammar** is a set of rules that describe the form of the elements that are valid according to the language syntax.

Here are some examples of tools for specifying grammars.

Bison and **Flex** (Levine 2009) are the most famous in the C context: from a high level specification of the syntactic structure (Bison) and lexical structure (Flex) of a language, they generate the parser and lexer in C, respectively. Bison is an implementation of Yacc (Yet Another Compiler-compiler, Brown et al. 1995), and there are variants for other languages as well, such as Racc for Ruby.

In the Java world, the most well-known is probably **ANTLR** (pronounced Antler, ANother Tool for Language Recognition) (Parr 2007). ANTLR allows the programmer to specify the grammar of the language in one single file (without separating the syntactic and lexical specifications in different files), and then it automatically generates the parser in Java.

Just to have an idea of what the specification of grammars looks like in ANTLR, here is the (simplified) grammar for an expression language for arithmetic expressions (with sum and multiplication):

```
expression
  : INT
  | expression '*' expression
  | expression '+' expression
;
```

Even if you do not understand all the details, it should be straightforward to get its meaning: an expression is either an integer literal, or (recursively) two expressions with an operator in between (either * or +).

From such a specification, you automatically get the Java code that will parse such expressions.

The Abstract Syntax Tree (AST)

Parsing a program is only the first stage in a programming language implementation. Once the program is checked as correct from the syntactic point of view, the implementation will have to do something with the elements of the program.

First of all, the overall correctness of a program cannot always be determined during parsing. One of the correctness checks that usually cannot be performed during parsing is type checking, that is, checking that the program is correct with respect to types. For instance, in Java, you cannot assign a string value to an integer variable, or, you can only assign instances of a variable's declared type or subclasses thereof.

Trying to embed type checking in a grammar specification could either make the specification more complex, or it could be simply impossible, since some type checks can be performed only when other program parts have already been parsed.

Type checking is part of the **semantic analysis** of a program. This often includes managing the **symbol table**, that is, for instance, handling the variables that are declared and that are visible only in specific parts of the program (think of fields in a Java class and their visibility in methods).

For these reasons, during parsing, we should also build a representation of the parsed program and store it in memory so that we can perform the semantic analysis on the memory representation without needing to parse the same text over and over again. A convenient representation in memory of a program is a tree structure called the **Abstract Syntax Tree** (or **AST**). The AST represents the abstract syntactic structure of the program. In this tree, each node represents a construct of the program.

Once the AST is stored in memory, the DSL implementation will not need to parse the program anymore, and it can perform all the additional semantic checks on the AST, and if they all succeed, it can use the AST for the final stage of the implementation, which can be the interpretation of the program or code generation.

In order to build the AST we need two additional things.

We need the code for representing the nodes of such a tree; if we are using Java, this means that we need to write some Java classes, typically one for each language construct. For instance, for the expression language we might write one class for the integer literal and one for the binary expression. Remember that since the grammar is recursive, we need a base class for representing the abstract concept of an expression. For example,

```
interface Expression { }

class Literal implements Expression {
  Integer value;
```

```
  // constructor and set methods...
}

class BinaryExpression implements Expression {
  Expression left, right;
  String operator;
  // constructor and set methods...
}
```

Then, we need to **annotate** the grammar specification with **actions** that construct the AST during the parsing; these actions are basically Java code blocks embedded in the grammar specification itself; the following is just a (simplified) example and it does not necessarily respect the actual ANTLR syntax:

```
expression:
  INT { $value = new Literal(Integer.parseInt($INT.text)); }
| left=expression '*' right=expression {
  $value = new BinaryExpression($left.value, $right.value);
  $value.setOperator("*");
}
| left=expression '+' right=expression {
  $value = new BinaryExpression($left.value, $right.value);
  $value.setOperator("+");
}
;
```

IDE integration

Even once you have implemented your DSL, that is, the mechanisms to read, validate, and execute programs written in your DSL, your work cannot really be considered finished.

Nowadays, a DSL should be shipped with good IDE support: all the IDE tooling that programmers are used to could really make the adoption of your DSL successful.

If your DSL is supported by all the powerful features in an IDE such as a syntax-aware editor, immediate feedback, incremental syntax checking, suggested corrections, auto-completion, and so on, then it will be easier to learn, use, and maintain.

In the following sections we will see the most important features concerning IDE integration; in particular, we will assume Eclipse as the underlying IDE (since Xtext is an Eclipse framework).

Syntax highlighting

The ability to see the program colored and formatted with different visual styles according to the elements of the language (for example, comments, keywords, strings, and so on) is not just "cosmetic".

First of all, it gives immediate feedback concerning the syntactic correctness of what you are writing. For instance, if string constants (typically enclosed in quotes) are rendered as red, and you see that at some point in the editor the rest of your program is all red, you may soon get an idea that somewhere in between you forgot to insert the closing quotation mark.

Moreover, colors and fonts will help the programmer to see the structure of the program directly, making it easier to visually separate the parts of the program.

Background parsing

The programming cycle consisting of writing a program with a text editor, saving it, shifting to the command line, running the compiler, and, in case of errors, shifting back to the text editor is surely not productive.

The programming environment should not let the programmer realize about errors too late; on the contrary, it should continuously check the program in the background while the programmer is writing, even if the current file has not been saved yet. The sooner the environment can tell the programmer about errors the better. The longer it takes to realize that there is an error, the higher the cost in terms of time and mental effort to correct.

Error markers

When your DSL parser and checker issue some errors, the programmer should not have to go to the console to discover such errors; your implementation should highlight the parts of the program with errors directly in the editor by underlining (for instance, in red) only the parts that actually contain the errors; it should also put some error markers (with an explicit message) on the left of the editor in correspondence to the lines with errors, and should also fill the **Problem** view with all these errors. The programmer will then have the chance to easily spot the parts of the program that need to be fixed.

Content assist

It is nice to have the editor propose some content when you write your programs in Eclipse. This is especially true when the proposed content makes sense in that particular program context. Content assist is the feature that automatically, or on demand, provides suggestions on how to complete the statement/expression the programmer just typed. For instance, when editing a Java file, after the keyword `new`, Eclipse proposes only Java class names as possible completions.

Again, this has to do with productivity; it does not make much sense to be forced to know all the syntax of a programming language by heart (especially for DSLs, which are not common languages such as Java), neither to know all the language's library classes and functions.

It is much better to let the editor help you with contextualized proposals.

In Eclipse the content assist is usually accessed with the keyboard shortcut *Ctrl + Space bar*.

Hyperlinking

Hyperlinking is a feature that makes it possible to navigate between references in a program; for example, from a variable to its declaration, or from a function call to where the function is defined. If your DSL provides declarations of any sort (for instance, variable declarations or functions) and a way to refer to them (for instance, referring to a variable or invoking a declared function), then it should also provide **Hyperlinking**: from a token referring to a declaration, it should be possible to directly jump to the corresponding declaration. This is particularly useful if the declaration is in a file different from the one being edited. In Eclipse this corresponds to pressing *F3* or using *Ctrl* + click.

This functionality really helps a lot if the programmer needs to inspect a specific declaration. For instance, when invoking a Java method, the programmer may need to check what that method actually does.

Hovering is a similar IDE feature: if you need some information about a specific program element, just hovering on that element should display a pop-up window with some documentation about that element.

Quickfixes

If the programmer made a mistake and your DSL implementation is able to fix it somehow, why not help the programmer by offering suggested quickfixes?

As an example, in the Eclipse Java editor, if you invoke a method that does not exist in the corresponding class, you are provided with some quickfixes (try to experiment with this); for instance, you are given a chance to fix this problem by actually creating such a method. This is typically implemented by a context menu available from the error marker.

In a test driven scenario this is actually a methodology. Since you write tests before the actual code to test, you can simply write the test that invokes a method that does not exist yet, and then employ the quickfix to let the IDE create that method for you.

Outline

If a program is big, it is surely helpful to have an **outline** of it showing only the main components; clicking on an element of the outline should bring the programmer directly to the corresponding source line in the editor.

Think about the outline view you get in Eclipse when editing a Java source file. The outline shows, in a compact form, all the classes and the corresponding methods of the currently edited Java file without the corresponding method bodies. Therefore, it is easy to have a quick overview of the contents of the file and to quickly jump to a specific class or method through the outline view.

Furthermore, the outline can also include other pieces of information such as types and structure that are not immediately understood by just looking at the program text. It is handy to have a view that is organized differently, perhaps sorted alphabetically to help with navigation.

Automatic build

In Eclipse, you have a Java project, and when you modify one Java file and save it, you know that Eclipse will automatically compile that file and, consequently, all the files that depend on the file you have just modified.

Having such functionality for your DSL will make its users happier.

Summarizing DSL implementation

In this section we briefly and informally introduced the main steps to implement a DSL.

The IDE tooling can be implemented on top of Eclipse, which already provides a comprehensive framework.

Indeed, all the features of the Eclipse Java editor (which is part of the project **JDT, Java Development Tools**) are based on the Eclipse framework, thus, you can employ all the functionalities offered by Eclipse to implement the same features for your own DSL.

Unfortunately, this task is not really easy: it certainly requires a deep knowledge of the internals of the Eclipse framework and lot of programming.

Finally, the parser will have to be connected to the Eclipse editing framework.

To make things a little bit worse, if you learned how to use all these tools (and this requires time) for implementing a DSL, when it comes to implement a new DSL, your existing knowledge will help you, but the time to implement the new DSL will still be huge.

All these learning and timing issues might push you to stick with XML, since the effort to produce a new DSL does not seem to be worthwhile. Indeed, there are many existing parsing and processing technologies for XML for different platforms that can be used, not to mention existing editors and IDE tooling for XML.

But what if there was a framework that lets you achieve all these tasks in a very quick way? What if this framework, once learned (yes, you cannot avoid learning new things), will let you implement new DSLs even quicker than the previous ones?

Enter Xtext

Xtext is an Eclipse framework for implementing programming languages and DSLs. It lets you implement languages quickly, and most of all, it covers all aspects of a complete language infrastructure, starting from the parser, code generator, or interpreter, up to a complete Eclipse IDE integration (with all the typical IDE features we discussed previously).

The really amazing thing about Xtext is that to start a DSL implementation, it only needs a grammar specification similar to ANTLR; it does not need to annotate the rules with actions to build the AST, since the creation of the AST (and the Java classes to store the AST) is handled automatically by Xtext itself. Starting from this specification, Xtext will automatically generate all the mechanisms sketched previously. It will generate the lexer, the parser, the AST model, the construction of the AST to represent the parsed program, and the Eclipse editor with all the IDE features!

Xtext comes with good and smart default implementations for all these aspects, and indeed most of these defaults will surely fit your needs. However, every single aspect can be customized by the programmer.

With all these features, Xtext is easy to use, it produces a professional result quickly, and it is even fun to use.

Installing Xtext

Xtext is an Eclipse framework, thus it can be installed into your Eclipse installation using the update site as follows:

```
http://download.eclipse.org/modeling/tmf/xtext/updates/composite/
releases
```

Just copy this URL into the dialog you get when you navigate to **Help | Install new software...** in the textbox **Work with** and press *Enter*; after some time (required to contact the update site), you will be presented with lots of possible features to install. The important features to install are **Xtend SDK 2.4.2** and **Xtext SDK 2.4.2**.

Alternatively, an Eclipse distribution for DSL developers based on Xtext is also available from the main Eclipse downloads page, `http://www.eclipse.org/downloads`, called Eclipse IDE for Java and DSL Developers.

> At the time of writing this book, the current version of Xtext was 2.4.2, and this is the version used in this book.

Let's try Xtext

Hopefully, by now you should be eager of seeing for yourself what Xtext can do! In this section we will briefly present the steps to write your first Xtext project and see what you get. Do not worry if you have no clue about most of the things you will see in this demo; they will be explained in the coming chapters.

1. Start Eclipse and navigate to **File | New | Project...**; in the dialog, navigate to the **Xtext** category and select **Xtext Project**.
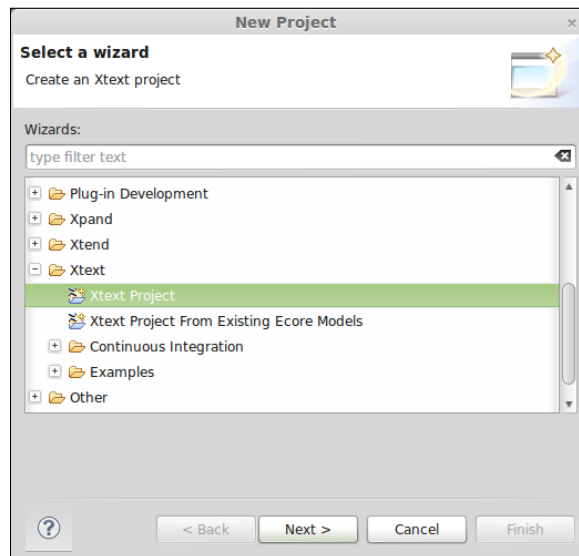
2. In the next dialog you can leave all the defaults, but uncheck the option **Create SDK feature project** (we will use the **Create SDK feature project** feature later in *Chapter 11*, *Building and Releasing*).
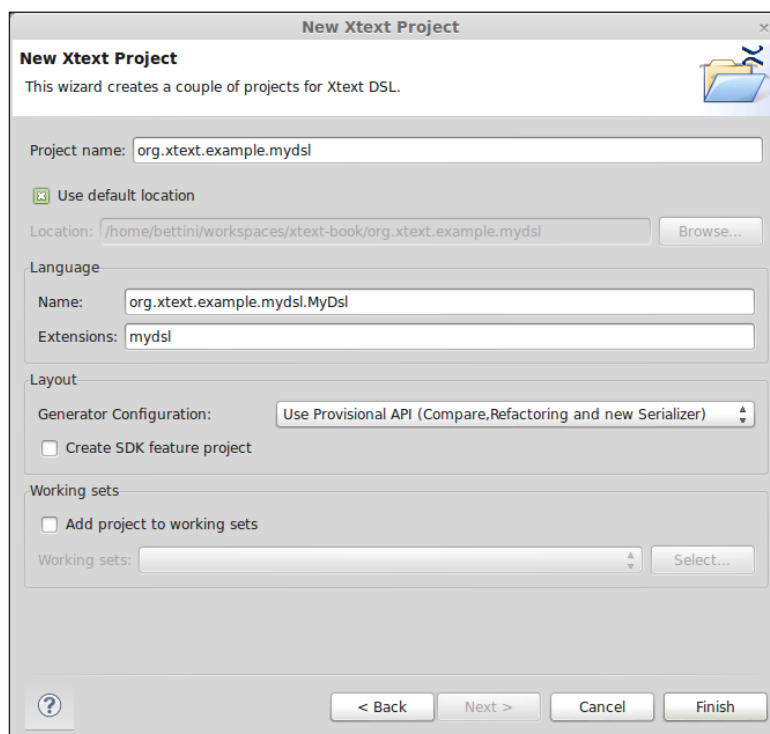
The wizard will create three projects and will open the file MyDsl.xtext ,which is the grammar definition of the new DSL we are about to implement. You do not need to understand all the details of this file's contents for the moment. But if you understood how the grammar definitions work from the examples in the previous sections, you might have an idea of what this DSL does. It accepts lines starting with the keyword Hello followed by an identifier, then followed by !.
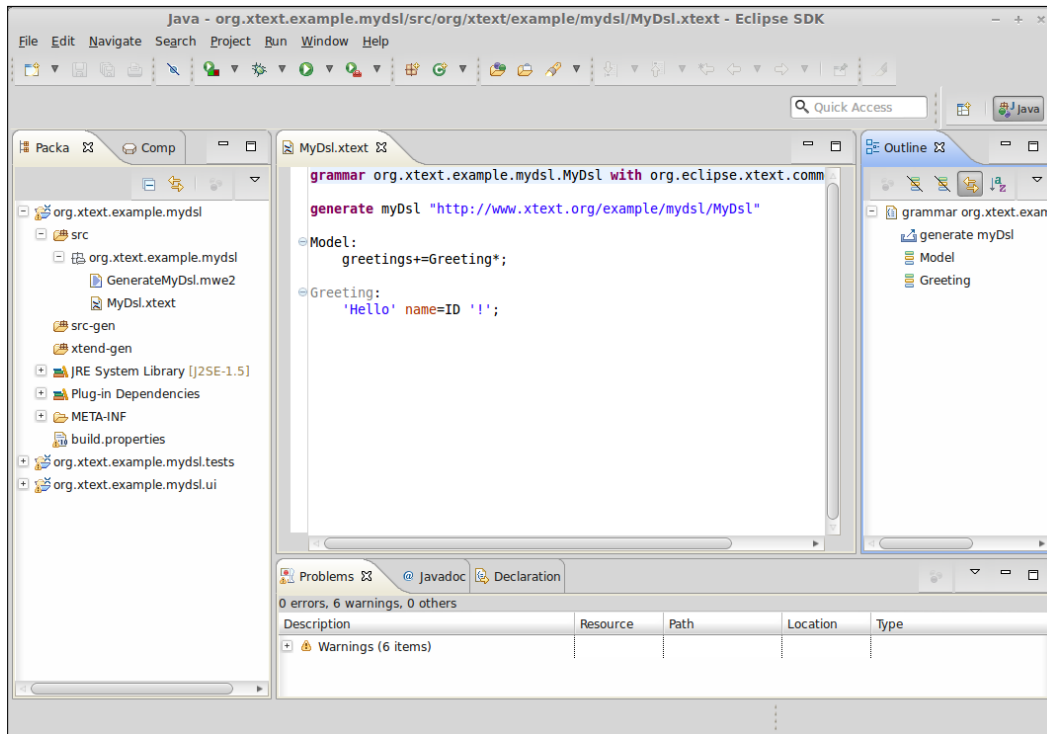
1. Now it is time to start the first Xtext generation, so navigate to the file MyDsl.xtext in the project org.xtext.example.mydsl, right-click on it, and navigate to **Run As | Generate Xtext Artifacts**. The output of the generation will be shown in the **Console** view. You will note that (only for the first invocation) you will be prompted with a question in the console:

```
*ATTENTION* It is recommended to use the ANTLR 3 parser generator
(BSD licence - http://www.antlr.org/license.html).Do you agree to
download it (size 1MB) from 'http://download.itemis.com/antlr-
generator-3.2.0.jar'? (type 'y' or 'n' and hit enter)
```

2. You should type y and press *Enter* so that this JAR will be downloaded and stored in your project once and for all (this file cannot be delivered together with Xtext installation: due to license problems, it is not compatible with the Eclipse Public License). Wait for that file to be downloaded, and once you read **Done** in the console, the code generation phase is finished, and you will note that the three projects now contain much more code. Of course, you will have to wait for Eclipse to build the projects.

3. Your DSL implementation is now ready to be tested! Since what the wizard created for you are Eclipse plug-in projects, you need to start a new Eclipse instance to see your implementation in action. Before you start the new Eclipse instance, you must make sure that the launch configuration has enough PermGen size, otherwise you will experience " out of memory " errors. You need to specify this VM argument in your launch configuration: `-XX:MaxPermSize=256m;` alternatively, you can simply use the launch configuration that Xtext created for you in your project `org.xtext. example.mydsl`, so right-click on that project and navigate to **Run As | Run Configurations...**; in the dialog, you can see **Launch Runtime Eclipse** under **Eclipse Application**; select that and click on **Run**.
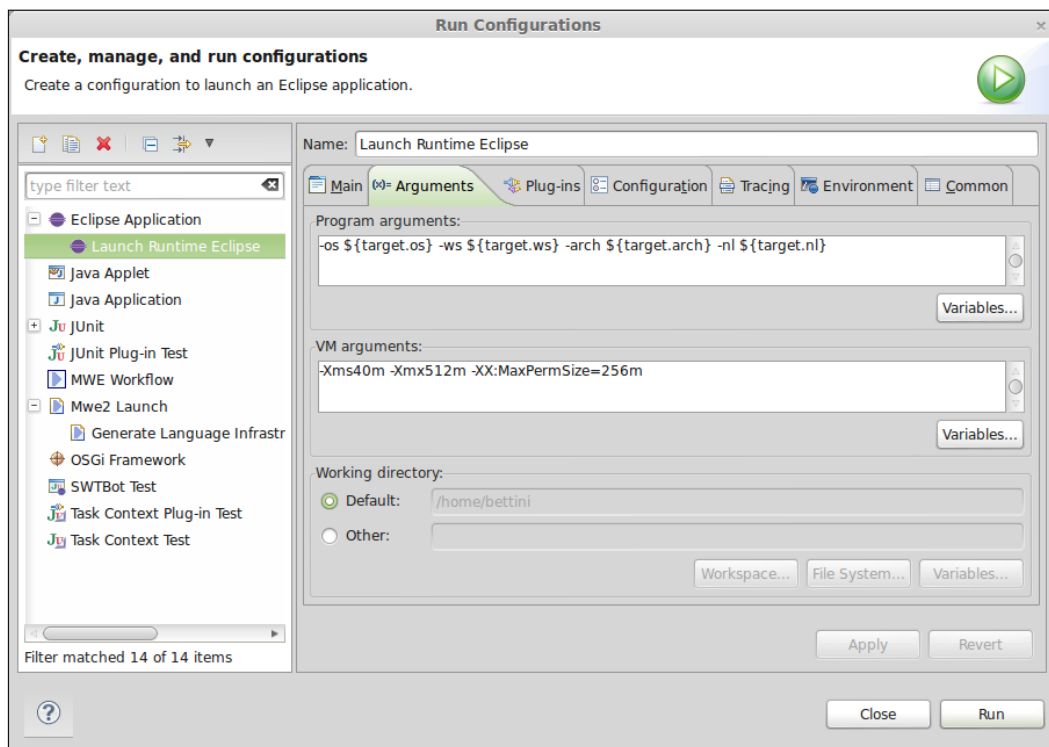
4. A new Eclipse instance will be run and a new workbench will appear. In this instance, your DSL implementation is available; so let's create a new **General** project (call it, for instance, sample). Inside this project create a new file; the name of the file is not important, but the file extension must be mydsl (remember that this was the extension we chose in the Xtext new project wizard). As soon as the file is created it will also be opened in a text editor and you will be asked to add the Xtext nature to your project. You should accept that to make your DSL editor work correctly in Eclipse.

5. Now try all the things that Xtext created for you! The editor features syntax highlighting (you can see that by default Xtext DSLs are already set up to deal with Java-like comments like // and /* */), immediate error feedback (with error markers only in the relevant parts of the file), outline view (which is automatically synchronized with the elements in the text editor), and code completion. All of these features automatically generated starting from a grammar specification file.
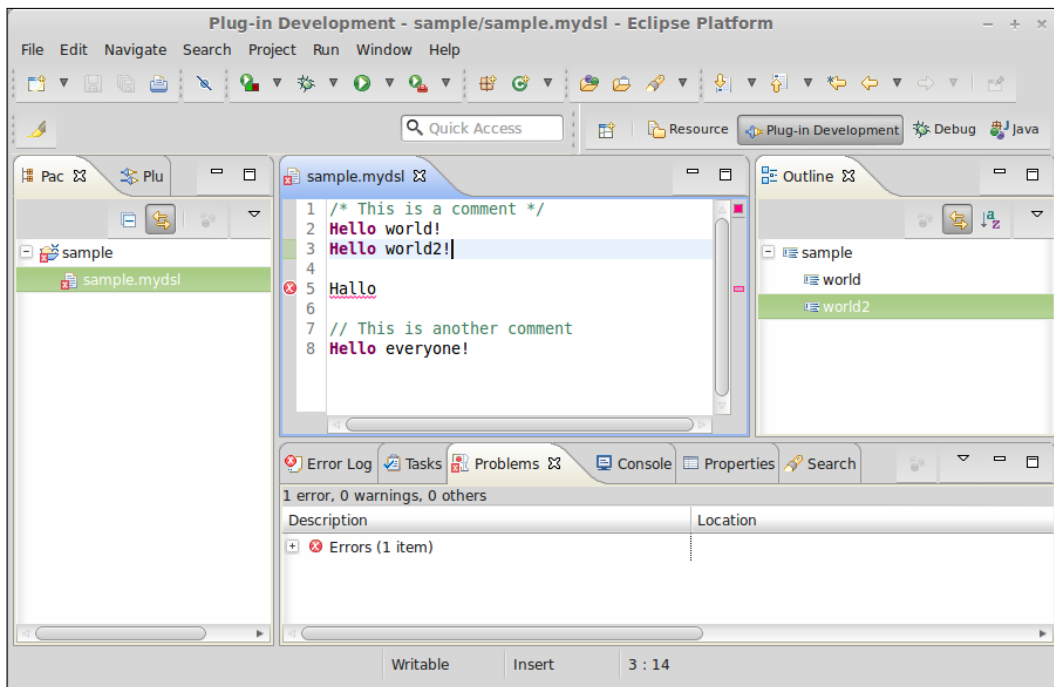
This short demo should have convinced you about the powerful features of Xtext (implementing the same features manually would require a huge amount of work). The result of the code generated by Xtext is so close to what Eclipse provides you for Java that your DSLs implemented in Xtext will be of high quality and will provide the users with all the IDE benefits.

The aim of this book

Xtext comes with some nice documentation; you can find such documentation in your Eclipse help system or online at `http://www.eclipse.org/Xtext/documentation.html`, where a PDF version of the whole documentation is available.

This book aims at being complementary to the official documentation, trying to give you enough information to start being productive in implementing DLSs with Xtext. This book will try to teach you some methodologies and best practices when using Xtext, filling some bits of information that are not present in the official documentation. Most chapters will have a tutorial nature and will provide you with enough information to make sure you understand what is going on. However, the official documentation should be kept at hand to learn more details about the mechanisms we will use throughout the book.

Summary

In this chapter we introduced the main concepts related to implementing a DSL, including IDE features.

At this point, you should also have an idea of what Xtext can do for you.

In the next chapter, we will use an uncomplicated DSL to demonstrate the main mechanisms and to get you familiar with the Xtext development workflow.

2
Creating Your First Xtext Language

In this chapter we will develop a DSL with Xtext and learn how the Xtext grammar language works. We will see the typical development workflow of programming with Xtext when we modify the grammar of the DSL. The chapter will also provide a small introduction to EMF (Eclipse Modeling Framework), a framework that Xtext relies upon to build the AST (Abstract Syntax Tree) of a program.

A DSL for entities

We will now implement a simple DSL to model entities, which can be seen as simple Java classes; each entity can have a super type entity (you can think of it as a Java superclass) and some attributes (similar to Java fields). This example is a variant of the domain model example that can be found in the Xtext documentation.

Creating the project

First of all, we will use the Xtext project wizard to create the projects for our DSL (we have already experimented with this at the end of *Chapter 1*, *Implementing a DSL*).

1. Start Eclipse and navigate to **File** | **New** | **Project...**. In the dialog navigate to the **Xtext** category and select **Xtext Project**.

2. In the next dialog you should specify the following names:

    ° **Project name:** `org.example.entities`
    ° **Name:** `org.example.entities.Entities`
    ° **Extensions:** `entities`
    ° Uncheck the option **Create SDK feature project** (we will use the **Create SDK feature project** only in *Chapter 11*, *Building and Releasing*)

The wizard will create three projects and it will open the file `Entities.xtext`, which is the grammar definition.

The main dialog of the wizard is shown in the following screenshot:
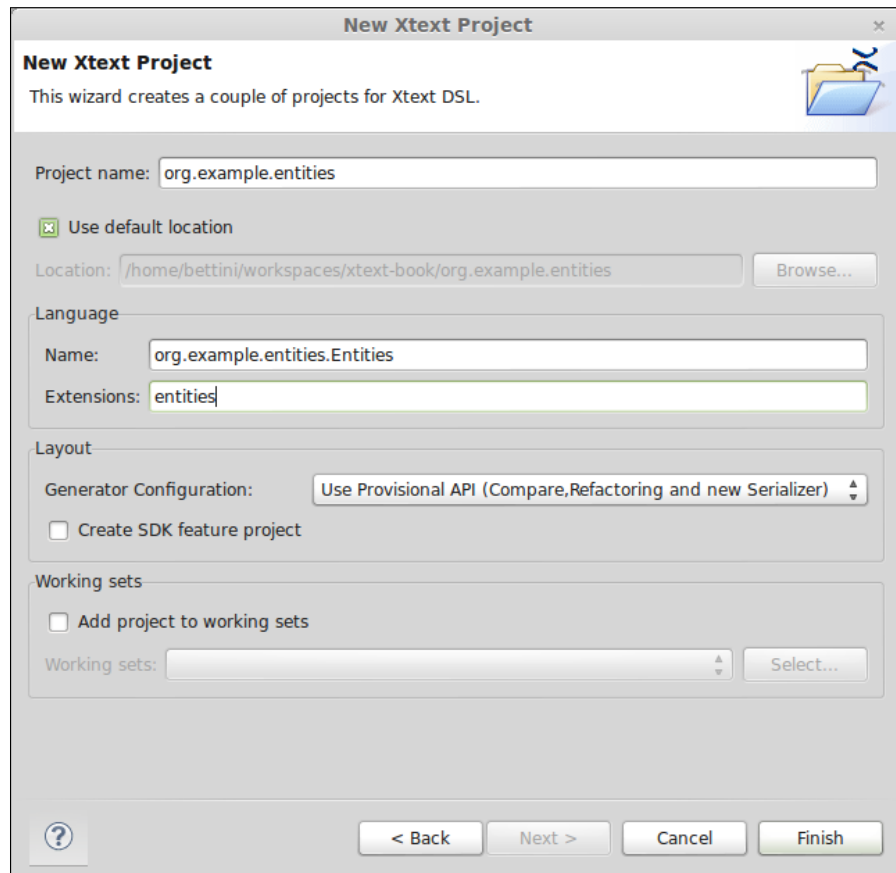
Xtext projects

The Xtext wizard generates three projects, and, in general, every DSL implemented in Xtext will have these three projects (with a name based on the **Project name** you specified in the wizard). In our example we have:

- `org.example.entities` is the main project that contains the grammar definition and all the runtime components that are independent from the UI
- `org.example.entities.tests` contains the unit tests

- `org.example.entities.ui` contains the components related to the UI (the Eclipse editor and features related to the user interface)

We will describe UI functionalities in *Chapter 6, Customizations* and unit tests in *Chapter 7, Testing*.

Modifying the grammar

As you may recall from *Chapter 1, Implementing a DSL*, a default grammar is generated by Xtext. In this section you will learn what this generated grammar contains, and we will modify it to contain the grammar for our Entities DSL. The generated grammar looks like the following:

```
grammar org.example.entities.Entities with org.eclipse.xtext.common.
Terminals

generate entities "http://www.example.org/entities/Entities"

Model:
  greetings+=Greeting*;

Greeting:
  'Hello' name = ID '!';
```

The first line declares the name of the language (and of the grammar), and it also corresponds to the fully qualified name of the `.xtext` file (the file is called `Entities.xtext`, and it is in the package `org.example.entities`).

The declaration of the grammar also states that it reuses the grammar `Terminals`, which defines the grammar rules for common things like quoted strings, numbers, and comments, so that in our language we will not have to define such rules. The grammar `Terminals` is part of the Xtext library; in *Chapter 12, Xbase* we will see another example of Xtext library grammar (the `Xbase` grammar).

The `generate` declaration defines some generation rules for EMF and we will discuss this later.

After the first two declarations, the actual rules of the grammar will be specified. For the complete syntax of the rules, you should refer to the official Xtext documentation (`http://www.eclipse.org/Xtext/documentation.html`). For the moment, all the rules we will write will have a name, a colon , the actual syntactic form accepted by that rule, and are terminated by a semicolon.

Now we modify our grammar as follows:

```
grammar org.example.entities.Entities with
  org.eclipse.xtext.common.Terminals

generate entities "http://www.example.org/entities/Entities"

Model: entities += Entity*;

Entity:
  'entity' name = ID ('extends' superType=[Entity])? '{'
    attributes += Attribute*
  '}'
;

Attribute:
  type=[Entity] array?=('[]')? name=ID ';';
```

The first rule in every grammar defines where the parser starts and the type of the root element of the model of the DSL, that is, of the Abstract Syntax Tree (AST). In this example, we declare that an Entities DSL program is a collection of `Entity` elements. This collection is stored in a `Model` object, in particular in a feature called `entities` (as we will see later, the collection is implemented as a list). The fact that it is a collection is implied by the operator `+=`. The star operator, `*`, states that the number of the elements (in this case `Entity`) is arbitrary; in particular, it can be any number >= 0. Therefore, a program can also be empty and contain no `Entity`.

> If we wanted our programs to contain at least one `Entity`, we should have used the operator + instead of *.

The shape of `Entity` elements is expressed in its own rule:

```
Entity:
  'entity' name = ID ('extends' superType=[Entity])? '{'
    attributes += Attribute*
  '}'
;
```

First of all, string literals (which in Xtext can be expressed with either single or double quotes) define keywords of the DSL. In this rule we have three keywords, namely `'entity'`, `'extends'`, `'{'`, and `'}'`.

Therefore, a valid entity declaration statement starts with the keyword `'entity'` followed by an ID; there is no rule defining ID in our grammar because that is one of the rules that we inherit from the grammar `Terminals`. If you are curious to know how an ID is defined, you can *Ctrl* + click on the ID in the Xtext editor and that will bring you to the grammar `Terminals`, where you can see that an ID starts with an optional `'^'` character, followed by a letter (`'a'..'z'|'A'..'Z'`), a `'$'` character, or an underscore `'_'` followed by any number of letters, `'$'` characters, underscores, and numbers (`'0'..'9'`):

```
'^'? ('a'..'z'|'A'..'Z'|'$'|'_')

  ('a'..'z'|'A'..'Z'|'$'|'_'|'0'..'9')*;
```

The optional `'^'` character is used to escape an identifier if there are conflicts with existing keywords. The parsed ID will be assigned to the feature `name` of the parsed `Entity` model element.

The `( )?` operator declares an optional part. Therefore, after the ID, you can write the keyword `'extends'` and the name of an `Entity`. This illustrates one of the powerful features of Xtext, that is, cross-references. In fact, what we want after the keyword `'extends'` is not just a name, but the name of an existing `Entity`. This can be expressed in the grammar using square brackets and the type we want to refer to. Xtext will automatically resolve the cross-reference by searching in the program for an element of that type (in our case an `Entity`) with the given name. If it cannot find it, it will automatically issue an error. Note that in order for this mechanism to work, the referred element must have a feature called `name`. As we will see in the following section, the automatic code completion mechanism will also take into consideration cross-references, thus proposing elements to refer to.

> By default, cross-references and their resolutions are based on the feature `name` and on an ID. This behavior can be customized as we will see in *Chapter 10, Scoping*.

Then, the curly brackets `'{' '}'` are expected and within them `Attribute` elements can be specified (recall the meaning of `+=` and `*`); these `Attribute` elements will be stored in the `attributes` feature of the corresponding `Entity` object.

```
Attribute:
  type=[Entity] array?=('[]')? name=ID ';';
```

The rule for Attribute requires an `Entity` name (as explained previously, this is a cross-reference) that will be stored in the `type` feature and a name for the attribute; attributes must also be terminated with `';'`. Note that after the type, an optional `'[]'` can be specified; in this case, the type of the attribute is considered an array type, and the feature `array` will be true. This feature is `Boolean` since we used the `?=` assign operator and after such an operator we specify an optional part.

Let's try the Editor

At the end of *Chapter 1*, *Implementing a DSL*, we saw how to run the Xtext generator; you should follow the same steps, but instead of right-clicking on the `.xtext` file and navigating to **Run As | Generate Xtext Artifacts**, we right-click on the `.mwe2` file (in our example it is `GenerateEntities.mwe2`) and navigate to **Run As | MWE2 Workflow**. (Remember to accept the request for downloading the ANTLR generator, as explained in *Chapter 1*, *Implementing a DSL*).

Before you start the new Eclipse instance, you must make sure that the launch configuration has enough `PermGen` size, otherwise you will experience "out of memory" errors.

You need to specify the following value as VM arguments in your launch configuration: `-XX:MaxPermSize=256m`.

You can also simply use the launch configuration that Xtext created for you in your `org.example.entities` project in the directory `.launch`; you might not be able to see that directory, since by default the workbench hides resources starting with a dot, so make sure to remove that filter in your workspace preferences. Alternatively, you can right-click on that project and navigate to **Run As | Run Configurations...**; in the dialog, you can see **Launch Runtime Eclipse** under **Eclipse Application**; select it and click on **Run**.

A new Eclipse instance will be run and a new workbench will appear; in this instance, our Entities DSL implementation is available. So let's create a new General project (call it, for instance, `sample`). Inside this project, create a new file; the name of the file is not important, but the file extension must be `entities` (remember that this was the extension we chose in the Xtext project wizard). As soon as the file is created, it will also be opened in a text editor, and you will be asked to add the **Xtext** nature to your project. You should accept that to make your DSL editor work correctly in Eclipse.

The editor is empty, but there is no error since an empty program is a valid Entities program (remember how the `Model` rule was defined with the operator `*`). If you access content assist (with *Ctrl + Space bar*), you will get no proposal; instead, the `entity` keyword is inserted for you. This is because the generated content assist is smart enough to know that in that particular program context there is only one valid thing to do: start with the keyword `entity`.

After that you get an error (refer to the following screenshot), since the entity definition is still incomplete (you can see that the syntax error tells you that an identifier is expected instead of the end of file):
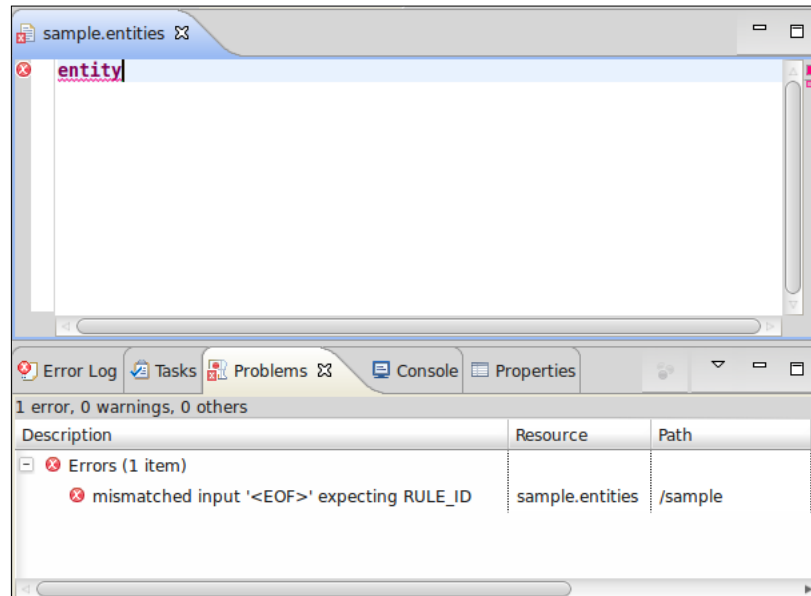
If you access the content assist again, you will get a hint that an identifier is expected (refer to the following screenshot), so let's write an identifier:
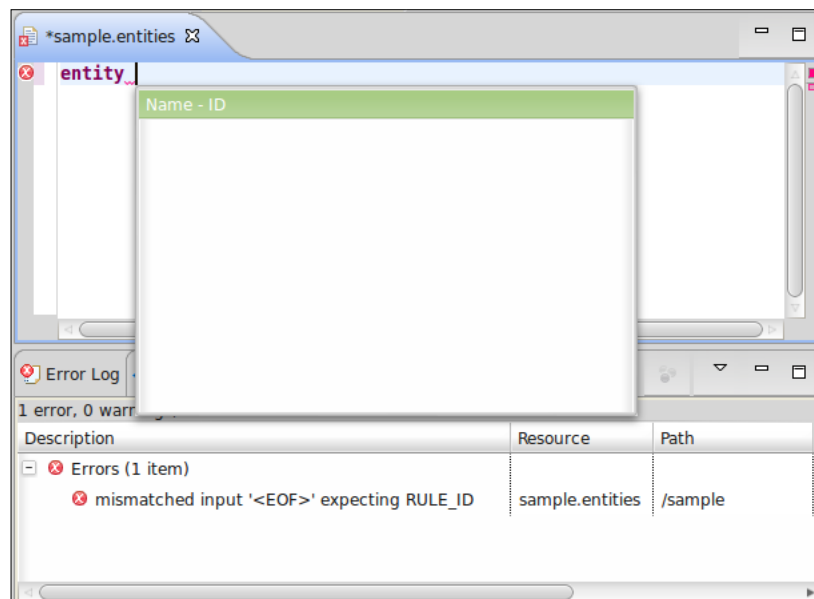
If you access the content assist after the identifier, you will see that you get two proposals (refer to the following screenshot). Again, the generated content assist knows that in that program context, you can continue either with an `'extends'` specification or with an open curly bracket.
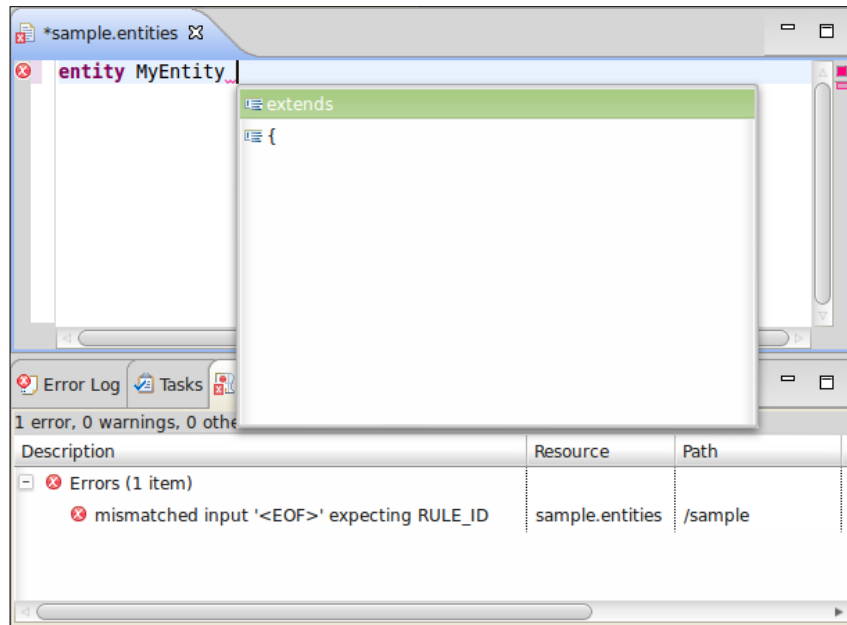
If you choose the open curly bracket, {, you will note some interesting things in the generated editor (refer to the following screenshot):

- The editor automatically inserts the corresponding closing curly bracket
- Inserting a newline between the brackets correctly performs indentation and moves the cursor to the right position
- The folding on the left of the editor is automatically handled
- The error marker turned gray, meaning that the problems in the current program are solved, but it has not been saved yet (saving the file makes the error marker go away and the **Problems** view becomes empty)
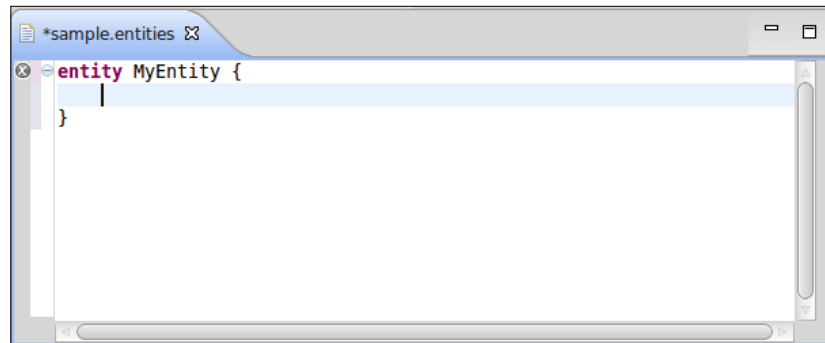
Continue experimenting with the editor; in particular, in the context where an entity reference is expected (that is, after the `extends` keyword or when declaring an attribute), you will see that the content assist will provide you with all the `Entity` elements defined in the current program.

> We should not allow an entity to extend itself; moreover, the hierarchy should be acyclic. However, there is no way to express these constraints in the grammar; these issues have to be dealt with by implementing a custom **Validator** (*Chapter 4, Validation*) or a custom **Scoping** mechanism (*Chapter 10, Scoping*).

We would also like to stress that all these functionalities, which are far from easy to implement manually, have been generated by Xtext starting from the grammar definition of our DSL.

The Xtext generator

Xtext uses the MWE2 DSL to configure the generation of its artifacts; the default generated `.mwe2` file already comes with good defaults, thus, for the moment, we will not modify it. However, it is interesting to know that by tweaking this file we can request the Xtext generator to generate support for additional features, as we will see later in this book.

During the MWE2 workflow execution, Xtext will generate artifacts related to the UI editor for your DSL, but most important of all, it will derive an ANTLR specification from the Xtext grammar with all the actions to create the AST while parsing. The classes for the nodes of the AST will be generated using the EMF framework (as explained in the next section).

The generator must be run after every modification to the grammar (the .xtext file). The whole generator infrastructure relies on the **Generation Gap Pattern** (Vlissides 1996). Indeed, code generators are fine, but when you have to customize the generated code: subsequent generations may overwrite your customizations. The Generation Gap Pattern deals with this problem by separating the code that is generated (and can be overwritten) from the code that you can customize (without the risk of being overwritten). In Xtext the generated code is placed in the source folder src-gen (this holds for all of the three projects); what is inside that source folder should never be modified, since on the next generation it will be overwritten. The programmer can instead safely modify everything in the source folder src. Indeed, on the first generation, Xtext will also generate a few stub classes in the source folder src to help the programmer with a starting point. These classes are never regenerated and can thus safely be edited without the risk of being overwritten by the generator. Some stub classes inherit from default classes from the Xtext library, while other stub classes inherit from classes which are in src-gen.

Most generated stub classes in the src folder are actually Xtend classes; the Xtend programming language will be introduced in the next chapter, thus, for the moment, we will not look at these stub classes.

There is one exception to the previously described generation strategy, which concerns the file plugin.xml (in the runtime and in the UI plug-ins): further Xtext generations will generate the file plugin.xml_gen in the root directory of your projects. It is up to you to check whether something has changed by comparing it with plugin.xml. In that case you should manually merge the differences. This can be easily done by using Eclipse: select the two files, right-click and navigate to **Compare With | Each Other...**, as illustrated in the following screenshot:
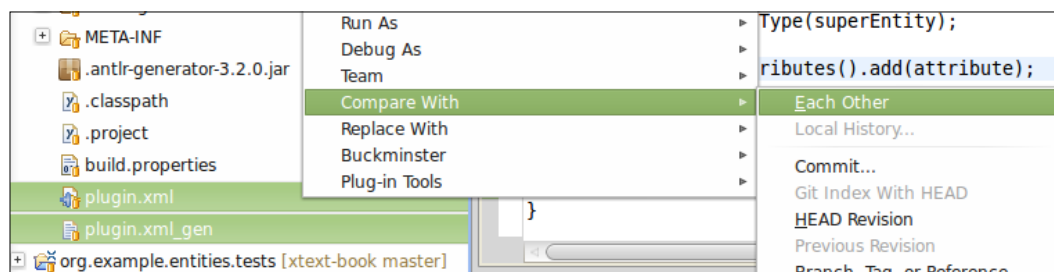
In general, checking the differences between plugin.xml and plugin.xml_gen is only needed either when modifying the .mwe2 file or when using a new version of Xtext (which can introduce new features).

Finally, after running the MWE2 workflow, since the grammar has changed, new EMF classes can be introduced or some existing EMF classes can be modified; thus, it is necessary to restart the other Eclipse instance where you are testing your editor.

The Eclipse Modeling Framework (EMF)

The Eclipse Modeling Framework (EMF) (Steinberg et al., 2008), `http://www.eclipse.org/modeling/emf`, provides code generation facilities for building tools and applications based on structured data models. Most of the Eclipse projects that in some way deal with modeling are based on EMF since it simplifies the development of complex software applications with its modeling mechanisms. The model specification (**metamodel**) can be described in XMI, XML Schema, UML, Rational Rose, or annotated Java. It is also possible to specify the metamodel programmatically using Xcore, which was implemented in Xtext. Typically, a metamodel is defined in the **Ecore** format, which is basically an implementation of a subset of UML class diagrams.

> Pay attention to the meta levels in this context: an Ecore model is a metamodel, since it is a model describing a model. Using the metamodel EMF produces a set of Java classes for the model. If you are not familiar with modeling technologies, you can think of a metamodel as a way of defining Java classes (that is, hierarchy relations, fields, method signatures, and so on). All Java classes generated by EMF are subclasses of `EObject`, which can be seen as the EMF equivalent of `java.lang.Object`. Similarly, `EClass` corresponds to `java.lang.Class` for dealing with introspection and reflection mechanisms.

Xtext relies on EMF for creating the AST, Abstract Syntax Tree, which we talked about in *Chapter 1*, *Implementing a DSL*. From your grammar specification, Xtext will automatically infer the EMF metamodel for your language. You can refer to the Xtext documentation for all the details about metamodel inference. For the moment, you can consider this simplified scenario: for each rule in your grammar, an EMF interface and class will be created with a field for each feature in the rule (together with a getter and setter). For instance, for the `Entity` rule, we will have the corresponding Java interface (and the corresponding implementation Java class):

```java
public interface Entity extends EObject {
  String getName();
  void setName(String value);
  Entity getSuperType();
  void setSuperType(Entity value);
  EList<Attribute> getAttributes();
}
```

Since these Java artifacts are generated, they are placed in the corresponding package in the `src-gen` folder.

You can have a look at the generated metamodel file `Entities.ecore` by opening it with the default EMF Ecore editor. Although you may not know the details of the description of a metamodel in EMF, it should be quite straightforward to grasp the meaning of it (refer to the following screenshot; in the screenshot, you can also see some expanded Java interfaces generated by EMF):
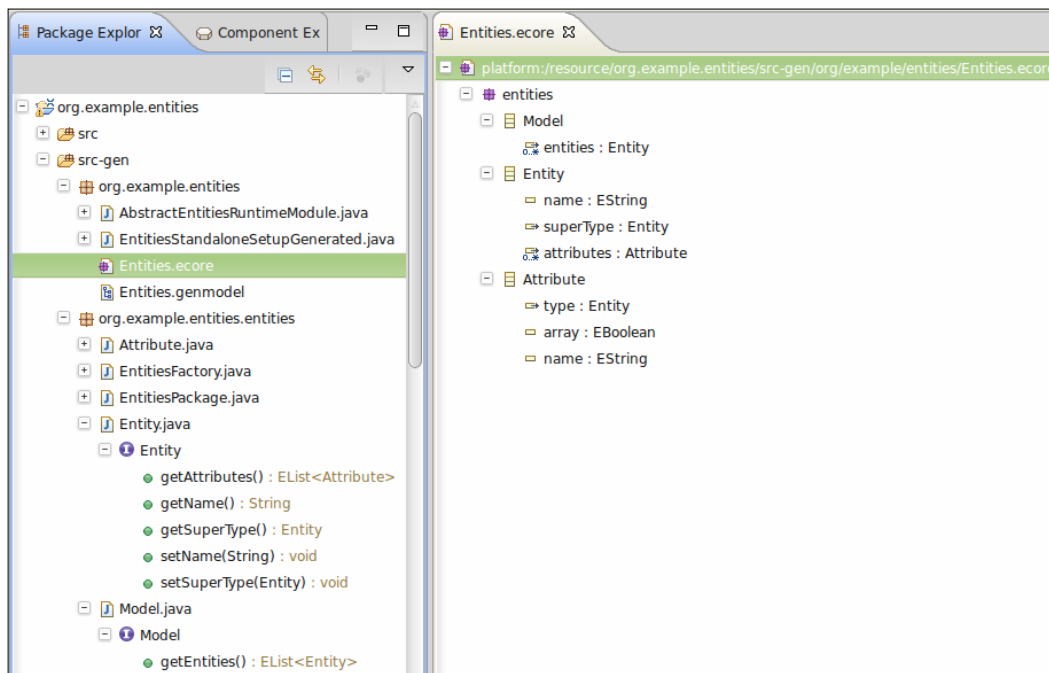
The inference of the metamodel and the corresponding EMF code generation is handled transparently and automatically by Xtext. However, Xtext can also use an existing metamodel that you maintain yourself, as detailed in the documentation (we will not use this mechanism in this book).

Since the model of your DSL programs are generated as instances of these generated EMF Java classes, a basic knowledge of EMF is required. As soon as you have to perform additional constraint checks for your DSL and to generate code, you will need to inspect this model and traverse it.

It is easy to use the generated Java classes since they follow conventions. In particular, instances of EMF classes must be created through a static factory (which results from EMF generation itself), thus there is no constructor to use; initialization of fields (that is, features) can be done with getters and setters. A collection in EMF is implemented as an `EList` interface (which is an extension of the standard library `List`). With only these notions in mind, it is easy to programmatically manipulate the model of your program. For instance, this Java snippet programmatically creates an Entities model that corresponds to an Entities DSL program:

```java
import org.example.entities.entities.Attribute;
import org.example.entities.entities.EntitiesFactory;
import org.example.entities.entities.Entity;
import org.example.entities.entities.Model;

public class EntitiesEMFExample {

  public static void main(String[] args) {
    EntitiesFactory factory = EntitiesFactory.eINSTANCE;

    Entity superEntity = factory.createEntity();
    superEntity.setName("MySuperEntity");

    Entity entity = factory.createEntity();
    entity.setName("MyEntity");
    entity.setSuperType(superEntity);

    Attribute attribute = factory.createAttribute();
    attribute.setName("myattribute");
    attribute.setArray(false);
    attribute.setType(superEntity);

    entity.getAttributes().add(attribute);

    Model model = factory.createModel();
    model.getEntities().add(superEntity);
    model.getEntities().add(entity);
  }
}
```

EMF is easy to learn, but as with any powerful tool, there is much to learn to fully master it. As hinted previously, it is widely used in the Eclipse world, thus you can consider it as an investment. Notably, the new Eclipse platform, called **e4**, uses an application model based on EMF, thus if you plan to develop RCP applications based on the new Eclipse e4, you will have to deal with EMF.

Improvements to the DSL

Now that we have a working DSL, we can do some improvements and modifications to the grammar.

After every modification to the grammar, as we said in the section *The Xtext generator*, we must run the MWE2 workflow so that Xtext will generate the new ANTLR parser and the updated EMF classes.

First of all, while experimenting with the editor, you might have noted that while

```
MyEntity[] myattribute;
```

is a valid sentence of our DSL, this one (note the spaces between the square brackets)

```
MyEntity[  ] myattribute;
```

produces a syntax error.

This is not good, since spaces should not be relevant in a DSL (although there are languages like Python and Haskell where spaces are indeed relevant).

The problem is due to the fact that in the `Attribute` rule, we specified `[]`, thus, no space is allowed between the square brackets; we can modify the rule as follows:

```
Attribute: type=[Entity] (array?='[' ']')? name=ID ';';
```

Since we split the two square brackets into two separate tokens, spaces between the brackets are allowed in the editor. Indeed, spaces are automatically discarded (unless they are explicit in the token definition).

We can further refine the array specification in our DSL by allowing an optional length:

```
Attribute:
  type=[Entity] (array ?='[' (length=INT)? ']')? name=ID ';';
```

There is no rule defining `INT` in our grammar: we inherit this rule from the grammar `Terminals`. As you can imagine, `INT` requires an integer literal, thus the `length` feature in our model will have an integer type as well. Since the `length` feature is optional (note the question mark), both the following attribute definitions will be valid sentences of our DSL:

```
MyEntity[ ] a;
MyEntity[10] b;
```

When the length is not specified, the `length` feature will hold the default integer value (`0`).

Dealing with types

The way we defined the concept of an attribute type is not conceptually correct, since the `array` feature is part of `Attribute` when it should be something that concerns only the `type` of `Attribute`.

We can then separate the concept of `AttributeType` in a separate rule (which will also result in a new EMF class in our model):

```
Attribute:
  type=AttributeType name=ID ';';

AttributeType:
  entity=[Entity] (array ?='[' (length=INT)? ']')?;
```

If you run the MWE2 workflow, you will note no difference in your DSL editor, but the metamodel for your AST has changed. For example, consider this part of the `EntitiesEMFExample` we showed previously:

```
Attribute attribute = factory.createAttribute();
attribute.setName("myattribute");
attribute.setArray(false);
attribute.setType(superEntity);
```

This is no longer valid Java code, and has to be changed as follows:

```
Attribute attribute = factory.createAttribute();
attribute.setName("myattribute");
AttributeType attributeType = factory.createAttributeType();
attributeType.setArray(false);
attributeType.setLength(10);
attributeType.setEntity(superEntity);
attribute.setType(attributeType);
```

As a further enhancement to our DSL, we would like to have some basic types: at the moment, only entities can be used as types. For instance, let us assume that our DSL provides three basic types: `string`, `int`, and `boolean`. Therefore, a basic type is represented by its literal representation. On the contrary, an entity type (the only type concept we have used up to now) is actually a reference to an existing `Entity`. Furthermore, we want to be able to declare arrays both of basic and of entity types.

For these reasons, the `array` feature still belongs to `AttributeType`, but we need to abstract over the element types; thus, we modify the grammar as follows:

```
AttributeType:
  elementType=ElementType (array ?='[' (length=INT)? ']')?;

ElementType:
  BasicType | EntityType;

BasicType:
  typeName=('string'|'int'|'boolean');

EntityType:
  entity=[Entity];
```

As you can see, we introduce a rule, `ElementType`, which in turn relies on two alternative rules (mutually exclusive): `BasicType` and `EntityType`. Alternative rules are separated using the pipe operator "|". Note that rules like `ElementType`, which basically delegates to other alternative rules, implicitly introduce an inheritance relation in the generated EMF classes; thus, both `BasicType` and `EntityType` inherit from `ElementType`. In the `BasicType` rule, the string feature `typeName` will contain the corresponding keyword entered in the program.

After running the MWE2 workflow, you can try your editor and see that now you can also use the three basic types; furthermore, in a context where a type specification is expected, the content assist (refer to the following screenshot) will present all possible element type alternatives (both entity types and basic types):
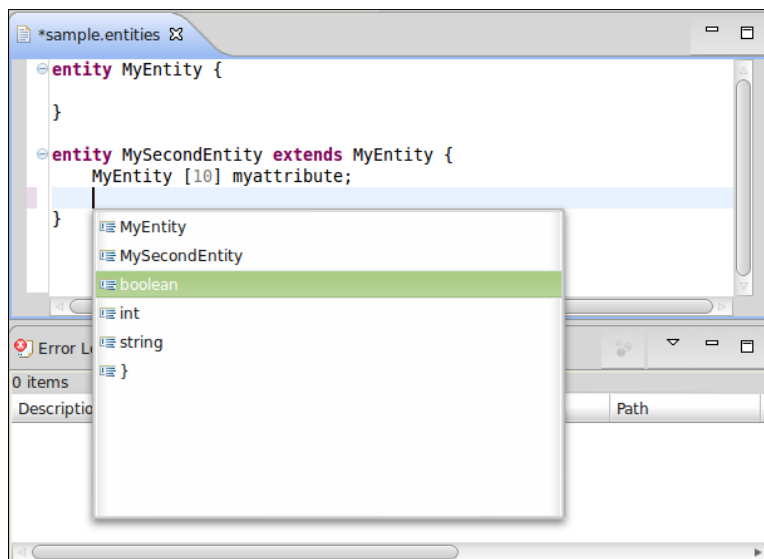
> In general, it is better not to rely on hardcoded alternatives in the grammar like the basic types we used previously. Instead, it would be better to provide a standard library for the DSL with some predefined types and functions. An application of this technique will be presented in *Chapter 10*, *Scoping*.

Now that our EMF model has changed again, we need to change our `EntitiesEMFExample` class accordingly as follows:

```
Attribute attribute = factory.createAttribute();
attribute.setName("myattribute");
AttributeType attributeType = factory.createAttributeType();
attributeType.setArray(false);
attributeType.setLength(10);
EntityType entityType = factory.createEntityType();
entityType.setEntity(superEntity);
attributeType.setElementType(entityType);
attribute.setType(attributeType);
```

If you reopen the generated `Entities.ecore` (refer to the following screenshot), you can see the current metamodel for the AST of our DSL (note the inheritance relations: **BasicType** and **EntityType** inherit from **ElementType**):
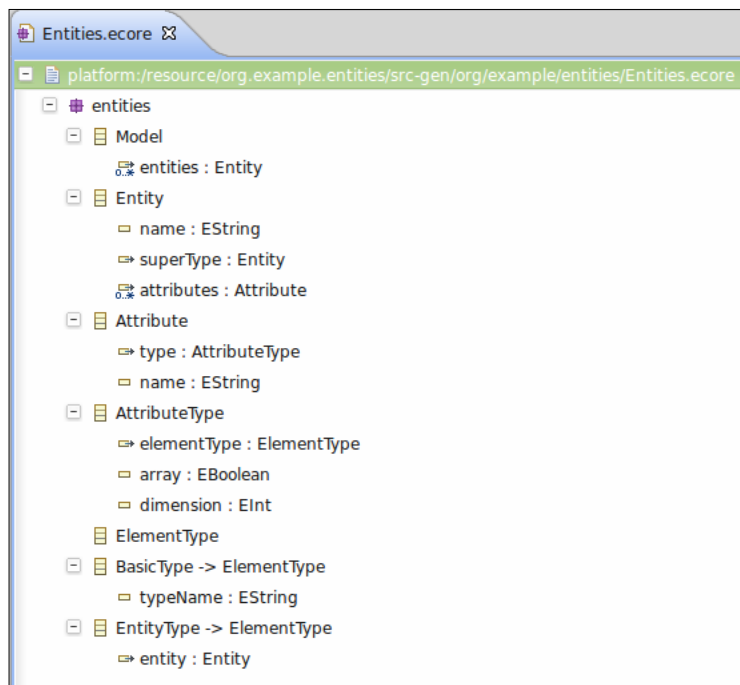
The preceding EMF metamodel can also be represented with a graphical notation following UML-like conventions (refer to the following diagram):
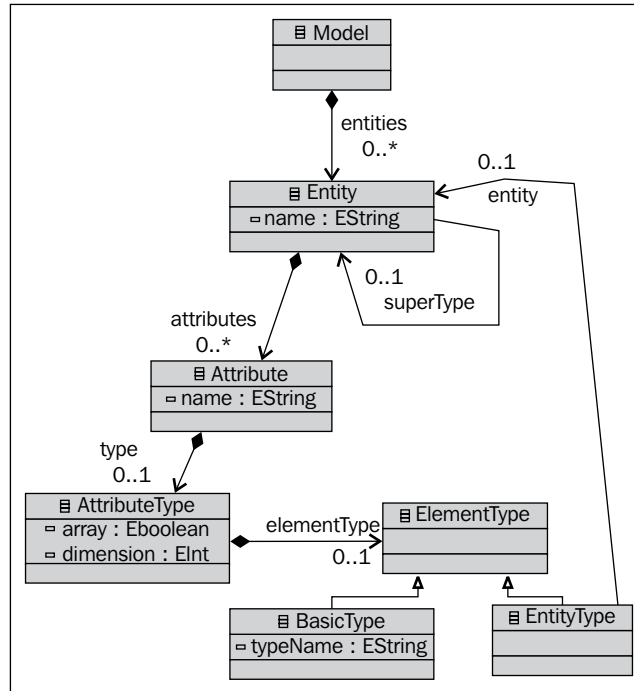
We now have enough features in the Entities DSL to start dealing with additional tasks typical of language implementation. We will be using this example DSL in the upcoming chapters.

Summary

In this chapter, you learned how to implement a simple DSL with Xtext and you saw that, starting from a grammar definition, Xtext automatically generates many artifacts for the DSL, including IDE tooling.

You also started to learn the EMF API that allows you to programmatically manipulate a model representing a program AST. Being able to programmatically access models is crucial to perform additional checks on a program that has been parsed and also to perform code generation, as we will see in the rest of the book.

In the next chapter, we will introduce the new programming language, Xtend (which is shipped with Xtext, and is implemented in Xtext itself): a Java-like general purpose programming language tightly integrated with Java that allows you to write much simpler and much cleaner programs. We will use Xtend in the rest of the book to implement all the aspects of languages implemented in Xtext.

3
The Xtend
Programming Language

In this chapter, we will introduce the Xtend programming language, a fully-featured Java-like language that is tightly integrated with Java. Xtend has a more concise syntax than Java and provides additional features such as type inference, extension methods, and lambda expressions, not to mention multi-line template expressions (which are useful when writing code generators). All the aspects of a DSL implemented in Xtext can be implemented in Xtend instead of Java, since it is easier to use and allows you to write better-readable code. Since Xtend is completely interoperable with Java, you can reuse all the Java libraries; moreover, all the Eclipse JDT (Java Development Tools) will work with Xtend seamlessly.

An introduction to Xtend

The Xtend programming language comes with very nice documentation, which can be found on its website, `http://www.eclipse.org/xtend`. We will give an overview of Xtend in this chapter, but we strongly suggest that you then go through the Xtend documentation thoroughly. Xtend itself is implemented in Xtext and it is a proof of concept of how involved a language implemented in Xtext can be.

We will use Xtend throughout this book to write all parts of a DSL implementation. Namely, we will use it to customize UI features, to write tests, to implement constraint checks, and to write code generators or interpreters for all the example DSLs we will develop in this book. In particular, starting with version 2.4, all the stub classes generated by Xtext for your DSL projects are Xtend classes by default (instead of Java, as in the previous versions).

You can still generate Java stub classes by customizing the MWE2 workflow, but in this book we will always use Xtend classes. Xtend, besides providing useful mechanisms for writing code generators (most of all, multi-line template expressions), also provides powerful features that make model visiting and traversing really easy, straightforward, and natural to read and maintain. Indeed, besides the grammar definition, for the rest of the time when implementing a DSL, you will have to visit the AST model. Xtend programs are translated into Java, and Xtend code can access all the Java libraries, thus Xtend and Java can coexist seamlessly.

Using Xtend in your projects

You can use Xtend in your Eclipse Java projects (both plain Java and plug-in projects).

In a plug-in project, you can right-click on your source folder and select **New | Xtend Class**; you will see that this wizard is similar to the standard New Java Class wizard, so you can choose the package, the class **Name**, **Superclass**, and **Interfaces** (refer the following screenshot):
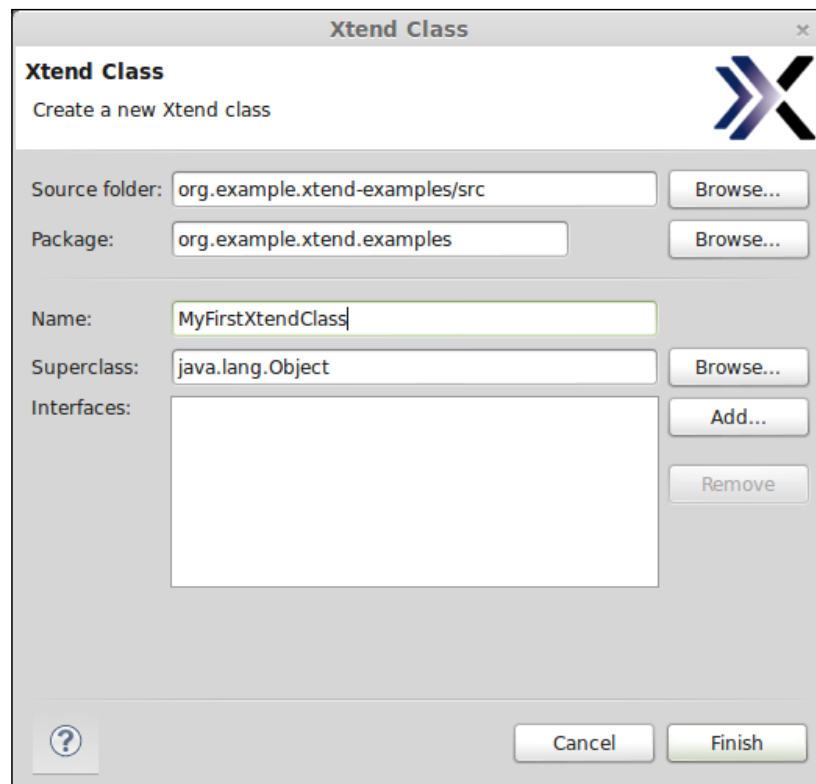
As soon as the class is created, you will get an error marker with the message "Mandatory library bundle 'org.eclipse.xtext.xbase.lib' 2.4.0 or higher not found on the classpath". You just need to use the quickfix "Add Xtend libs to classpath" and the required Xtend bundles will be added to your project's dependencies.

A new source folder will be created in your plug-in project, `xtend-gen`, where the Java code corresponding to your Xtend code will be automatically generated (using the building infrastructure of Eclipse) as soon as you save an `.xtend` file. Just like `src-gen` created by the Xtext generator (as seen in the previous chapter), the files in `xtend-gen` must not be modified by the programmer, since they will be overwritten by the Xtend compiler.

> The folder `xtend-gen` is not automatically added to the build source folders of your plug-in project, and therefore, you should add it manually in your `build.properties` file (the file has a warning marker, and the editor will provide you with a quickfix to add that folder). This is requested only for plug-in projects.

You can use the same steps to create an Xtend class in a plain Java project (again, you will have to use the quickfix to add the Xtend libraries to the classpath). Of course, in this case, there is no `build.properties` to adjust.

> Starting from version 2.4.0, Xtext automatically generates Xtend stub classes for your DSL (instead of the Java ones), thus, the runtime and UI plug-in projects are already setup to use Xtend and its libraries. However, this does not hold for the `tests` plug-in project; thus, when starting to write Xtend classes in the `tests` plug-in project (as we will see in *Chapter 7, Testing*), you will need to perform the setup steps described in this section.

Xtend – a better Java with less "noise"

Xtend is a statically typed language and it uses the Java type system (including Java generics). Thus Xtend and Java are completely interoperable.

Most of the linguistic concepts of Xtend are very similar to Java, that is, classes, interfaces, and methods. Moreover, Xtend supports most of the features of Java annotations. One of the goals of Xtend is to have a less "noisy" version of Java; indeed, in Java, some linguistic features are redundant and only make programs more verbose.

Let's write a "Hello World" program in Xtend:

```
package org.example.xtend.examples

class XtendHelloWorld {
  def static void main(String[] args) {
    println("Hello World")
  }
}
```

You can see that it is similar to Java, though there are some differences. First of all, the missing semicolons ; are not mistakes; in Xtend, they are not required (though they can be used). All method declarations start with either `def` or `override` (explained later in the chapter). Methods are `public` by default.

Note that the editor works almost the same as the one provided by JDT (for example, see the Outline view for the Xtend class in the following screenshot). You may also want to have a look at the generated Java class in the `xtend-gen` folder corresponding to the Xtend class (refer the following screenshot):
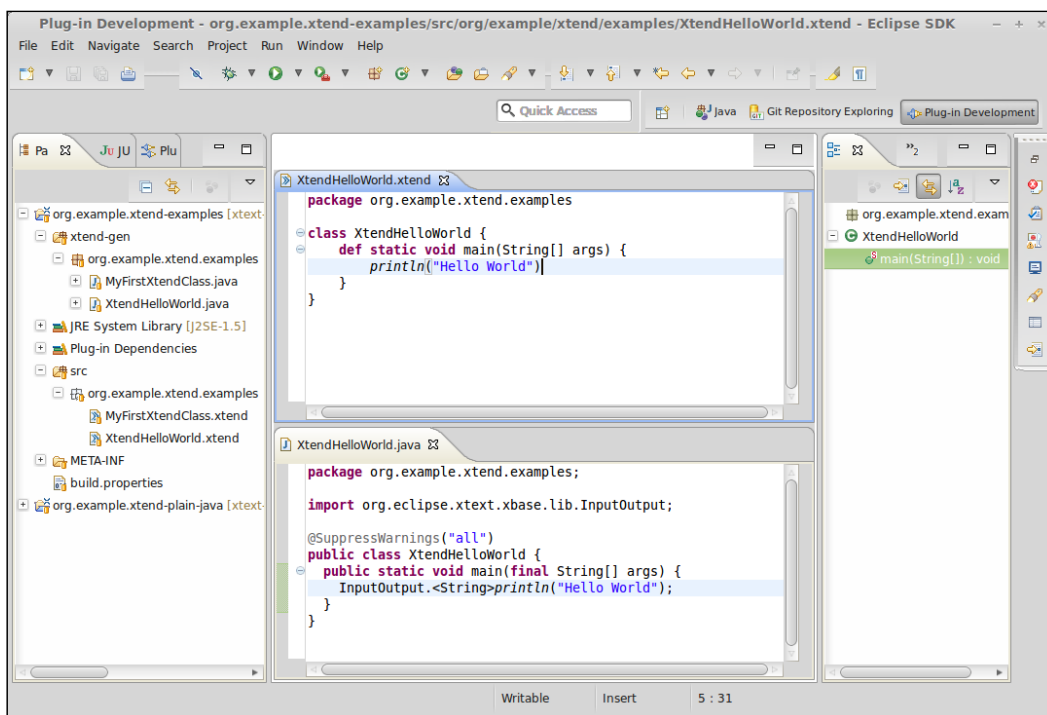
Although it is usually not required to see the generated Java code, it might be helpful, especially when starting to use Xtend, to see what is generated in order to learn Xtend's new constructs. Instead of manually opening the generated Java file, you can open the Xtend **Generated Code** view; the contents of this view will show the generated Java code, in particular, the code corresponding to the section of the Xtend file you are editing (refer the following screenshot). This view will be updated when the Xtend file is saved.
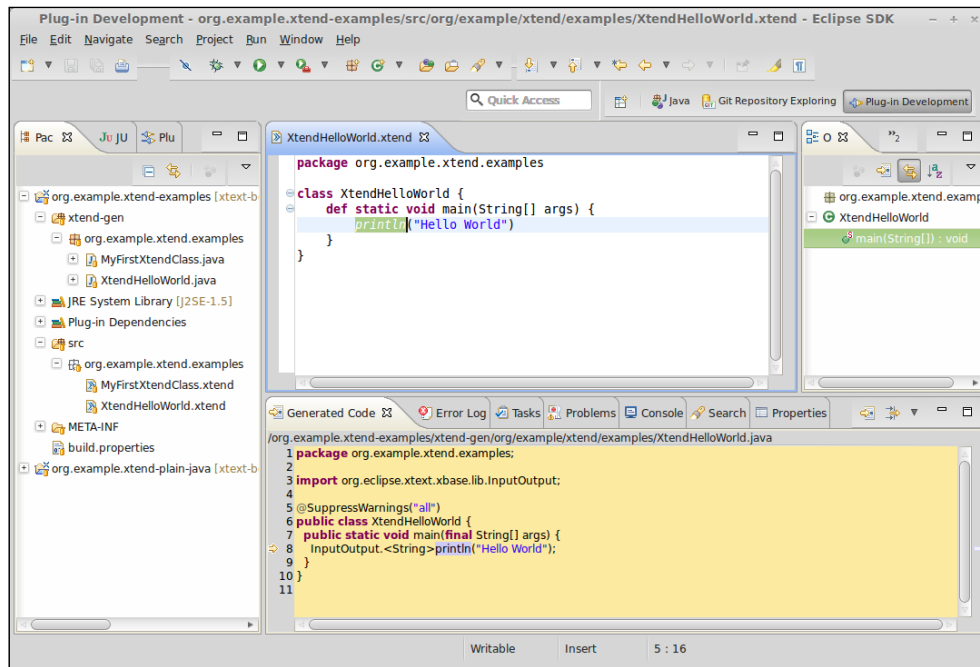
Here are some more Xtend examples:

```
class MyFirstXtendClass {
  val s = 'my field' // final field
  var myList = new LinkedList<Integer> // non final field

  def bar(String input) {
    var buffer = input
    buffer == s || myList.size > 0
    // the last expression is the return expression
  }
}
```

Fields and local variables are declared using `val` (for final fields and variables) and `var` (for non-final fields and variables). Fields are private by default.

You may have noticed that we wrote == for comparing two Java strings. This is usually one of the common mistakes in Java, since you will then compare the object references, not their actual values. However, Xtend handles operators transparently and correctly, thus == actually compares the values of the strings (indeed, it maps to the method equals).

Xtend provides some "syntactic sugar" (that is, syntax that is designed to write code which is easier to read) for getter and setter methods, so that instead of writing, for example, o.getName(), you can simply write o.name; similarly, instead of writing o.setName("..."), you can simply write o.name = "...". The same convention applies for boolean fields according to JavaBeans conventions (where the getter method starts with is instead of get). Similar syntactic sugar is available for method invocations, so that when a method has no parameter, the parenthesis can be avoided. Therefore, in the preceding code, myList.size corresponds to myList.size().

The preceding code also shows other important features of Xtend:

- **Type inference**: when the type of a variable can be inferred (for example, from its initialization expression), you are not required to specify it.
- **Everything is an expression**: in Xtend there are no statements, everything is an expression; in a method body, the last expression is the return expression (note the absence of return, although a return expression can be explicitly specified).

The declaration of a method's return type is not required: it is inferred from the returned expression (in this example it is boolean).

> Text hovering the Xtend elements in the editor will give you information about the inferred types.
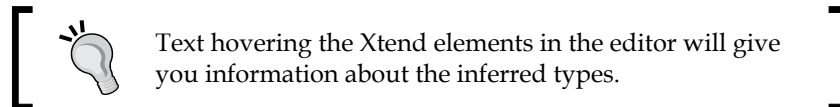
Note that the types of method parameters must always be specified.

Access to static members (fields and methods) of classes in Xtend must be done using the operator ::, instead of . that is used only for non-static members (differently from Java, where . is used for both), for example:

```
import java.util.Collections
class StaticMethods {
  def static void main(String[] args) {
    val list = Collections::emptyList
    System::out.println(list)
  }
}
```

Access to inner classes/interfaces of a Java class is done by using $, for example, given this Java class:

```
public class MyJavaClass {
  interface MyInnerInterface {
    public static String s = "s";
  }
}
```

We can access the inner interface in Xtend using the following syntax:

```
MyJavaClass$MyInnerInterface
```

and, accordingly, the static field can be accessed as follows:

```
MyJavaClass$MyInnerInterface::s
```

> In Xtend 2.4.2, static members and inner types are accessible with the dot operator as well. Therefore, the preceding line can also be written as follows: `MyJavaClass.MyInnerInterface.s`.

References to a Java class, that is, a **type literal** (which in Java you refer to by using the class name followed by .class), are expressed with typeof(class name), for example:

```
typeof(Entity) // corresponds to Entity.class in Java
```

> In Xtend 2.4.2, type literals can also be specified by their simple name: instead of `typeof(Entity)`, you can simply write `Entity`.

Xtend is stricter concerning method overriding: if a subclass overrides a method, it must explicitly define that method with override instead of def, otherwise a compilation error is raised. This should avoid accidental method overrides (that is, you did not intend to provide an overridden version of a method of a superclass). Even more importantly, if the method that is being overridden later is removed, you would want to know why your method may not ever be called.

In Xtend (and in general, by default, in any DSL implemented with Xtext using the default terminals grammar), strings can be specified both with single and double quotes. This allows the programmer to choose the preferred format depending on the string contents so that quotes inside the string do not have to be escaped, for example:

```
val s1 = "my 'string'"
val s2 = 'my "string"'
```

Escaping is still possible using the backslash character \ as in Java.

Extension methods

Extension methods is a syntactic sugar mechanism that allows you to add new methods to existing types without modifying them; instead of passing the first argument inside the parentheses of a method invocation, the method can be called with the first argument as its receiver. It is as if the method was one of the argument type's members.

For example, if `m(Entity)` is an extension method, and `e` is of type `Entity`, you can write `e.m()` instead of `m(e)`, even though `m` is not a method defined in `Entity`.

Using extension methods often results in a more readable code, since method calls are chained; for example, `o.foo().bar()` rather than nested, for example: `bar(foo(o))`.

Xtend provides several ways to make methods available as extension methods, as described in this section.

Xtend provides a rich runtime library with several utility classes and static methods. These static methods are automatically available in Xtend code so that you can use all of them as extension methods.

Of course, the editor also provides code completion for extension methods so that you can experiment with the code assistant. These utility classes aim at enhancing the functionalities of standard types and collections.
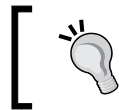
> Extension methods are highlighted in orange in the Xtend editor.

For example, you can write:

```
"my string".toFirstUpper
```

instead of:

```
StringExtensions.toFirstUpper("my string")
```

Similarly, you can use some utility methods for collections, for example, `addAll` as in the following code:

```
val list = new ArrayList<String>
list.addAll("a", "b", "c")
```

You can also use static methods from existing Java utility classes (for example, `java.util.Collections`) as extension methods by using a **static extension import** in an Xtend source file, for example:

```
import static extension java.util.Collections.*
```

In that Xtend file, all the static methods of `java.util.Collections` will then be available as extension methods.

Methods defined in an Xtend class can automatically be used as extension methods in that class; for example:

```
class ExtensionMethods {
  def myListMethod(List<?> list) {
    // some implementation
  }

  def m() {
    val list = new ArrayList<String>
    list.myListMethod
  }
}
```

Finally, by adding the "`extension`" keyword to a field, a local variable, or a parameter declaration, its instance methods become extension methods in that class, code block, or method body, respectively. For example, assume you have this class:

```
class MyListExtensions {

  def aListMethod(List<?> list) {
    // some implementation
  }

  def anotherListMethod(List<?> list) {
    // some implementation
  }
}
```

and you want to use its methods as extension methods in another class, `C`. Then, in `C`, you can declare an extension field (that is, a field declaration with the `extension` keyword) of type `MyListExtensions,` and in the methods of `C,` you can use the methods declared in `MyListExtensions` as extension methods:

```
class C {

  extension MyListExtensions e = new MyListExtensions

  def m() {
    val list = new ArrayList<String>
    list.aListMethod
    list.anotherListMethod
  }
```

You can see that the two methods of `MyListExtensions` are used as extension methods in `C`. Indeed, the two method invocations inside the method `m` are equivalent to:

```
e.aListMethod(list)
e.anotherListMethod(list)
```

As mentioned earlier, you can achieve the same goal by adding the keyword `extension` to a local variable:

```
def m() {
  val extension MyListExtensions e = new MyListExtensions
  val list = new ArrayList<String>
  list.aListMethod
  list.anotherListMethod
}
```

or to a parameter declaration:

```
def m(extension MyListExtensions e) {
  val list = new ArrayList<String>
  list.aListMethod
  list.anotherListMethod
}
```

When declaring a field with the keyword `extension`, the name of the field is optional. The same holds true when declaring a local variable with the keyword `extension`.

As we hinted previously, the static methods of the classes of the Xtend library are automatically available in Xtend programs. Indeed, the static methods can be used independently of the extension method's mechanisms. The `println` method in the first `XtendHelloExample` class is indeed a static method from an Xtend utility class. Many utility methods are provided for dealing with collections; in particular, instead of writing:

```
val list = new ArrayList<String>
list.addAll("a", "b", "c")
```

we can simply write:

```
val list = newArrayList("a", "b", "c")
```

By using these methods, you make full use of the type inference mechanism of Xtend.

The implicit variable – it

You know that in Java, the special variable `this` is implicitly bound in a method to the object on which the method was invoked; the same holds true in Xtend. However, Xtend also introduces another special variable `it`. While you cannot declare a variable or parameter with the name `this`, you are allowed to do so using the name `it`. If in the current program context a declaration for `it` is available, then all the members of that variable are implicitly available without using the `.` (just like all the members of `this` are implicitly available in an instance method), for example:

```
class ItExamples {
  def trans1(String it) {
    toLowerCase // it.toLowerCase
  }

  def trans2(String s) {
    var it = s
    toLowerCase // it.toLowerCase
  }
}
```

This allows you to write much more compact code.

Lambda expressions

A **lambda expression** (or **lambda** for short) defines an anonymous function. Lambda expressions are first class objects that can be passed to methods or stored in a variable for later evaluation.

Lambda expressions are typical of functional languages that existed long before object-oriented languages were designed. Therefore, as a linguistic mechanism, they are so old that it is quite strange that Java has not provided them from the start (they are planned to be available in Java 8). In Java, lambda expressions can be simulated with anonymous inner classes, as in the following example:

```
import java.util.*;
public class JavaAnonymousClasses {
  public static void main(String[] args) {
    List<String> strings = new ArrayList<String>();
    ...
    Collections.sort(strings,  new Comparator<String>() {
      public int compare(String left, String right) {
        return left.compareToIgnoreCase(right);
      }
    });
  }
}
```

In the preceding example, the sorting algorithm would only need a function implementing the comparison; thus, the ability to pass anonymous functions to other functions would make things much easier (anonymous inner classes in Java are often used to simulate anonymous functions).

Indeed, most uses of anonymous inner classes employ interfaces (or abstract classes) with only one method (for this reason, they are also called **SAM types – Single Abstract Method**); this should make it even more evident that these inner classes aim to simulate lambda expressions.

Xtend supports lambda expressions: they are declared using square brackets `[]`; parameters and the actual body are separated by a pipe symbol, `|`. The body of the lambda is executed by calling its `apply` method and passing the needed arguments.

The following code defines a lambda expression that is assigned to a local variable, taking a string and an integer as parameters and returning the string concatenation of the two. It then evaluates the lambda expression passing the two arguments:

```
val l = [ String s, int i | s + i ]
println(l.apply("s", 10))
```

Xtend also introduces types for lambda expressions (**function types**); parameter types (enclosed in parentheses) are separated from the return type by the symbol `=>` (of course, generic types can be fully exploited when defining lambda expression types). For example, the preceding declaration could have been written with an explicit type as follows:

```
val (String, int)=>String l = [ String s, int i | s + i ]
```

Recall that Xtend has powerful type inference mechanisms: variable type declarations can be omitted when the context provides enough information. In the preceding declaration, we made the type of the lambda expression explicit, thus the types of parameters of the lambda expression are redundant since they can be inferred:

```
val (String, int)=>String l = [ s, i | s + i ]
```

Function types are useful when declaring methods that take a lambda expression as a parameter (remember that the types of parameters must always be specified), for example:

```
def execute((String, int)=>String f) {
  f.apply("s", 10)
}
```

We can then pass a lambda expression as an argument to this method. When we pass a lambda as an argument to this method, there is enough information to fully infer the types of its parameters, which allows us to omit these declarations:

```
execute([s, i | s + i])
```

A lambda expression also captures the current scope; all final local variables and all parameters that are visible at definition time can be referred to from within the lambda expression's body. This is similar to Java anonymous inner classes that can access surrounding `final` variables and `final` parameters.

> In Xtend, method parameters are always automatically `final`.

Indeed, under the hood, Xtend generates Java inner classes for lambda expressions (and it will be Java 8 compatible in the future). The lambda expression is "closed" over the environment in which it was defined: the referenced variables and parameters of the enclosing context are captured by the lambda expression. For this reason, lambda expressions are often referred to as **closures.**

For example, consider the following code:

```
package org.example.xtend.examples
class LambdaExamples {
  def static execute((String,int)=>String f) {
    f.apply("s", 10)
  }
  def static void main(String[] args) {
    val c = "aaa"
    println(execute([ s, i | s + i + c ])) // prints s10aaa
  }
}
```

You can see that the lambda expression uses the local variable `c` when it is defined, but the value of that variable is available even when it is evaluated.
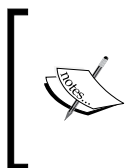
> Formally, lambda expressions are a linguistic construct, while closures are an implementation technique. From another point of view, a lambda expression is a function literal definition while a closure is a function value. However, in most programming languages and in the literature, the two terms are often used interchangeably.

Although function types are not available in Java, Xtend can automatically perform the required conversions; in particular, Xtend can automatically deal with Java SAM (Single Abstract Method) types: if a Java method expects an instance of a SAM type, in Xtend, you can call that method by passing a lambda (Xtend will perform all the type checking and conversions). This is a further demonstration of how Xtend is tightly integrated with Java.

Therefore, the Java example using `java.util.Collections.sort` passing an inner class can be written in Xtend as follows (the code is much more compact when using a lambda):

```
val list = newArrayList("Second", "First", "Third")
Collections::sort(list,
  [ arg0, arg1 | arg0.compareToIgnoreCase(arg1) ])
```

Again, note how Xtend infers the type of the parameters of the lambda.

Xtend also supports loops; in particular, in `for` loops, you can use the type inference and avoid writing the type (for example, `for (s : list)`). However, if you get familiar with lambdas, you will discover that you will tend not to use loops most of the time. In particular, in Java, you typically use loops to find something in a collection; with Xtend lambdas (and all the utility methods of the Xtend library that can be automatically used as extension methods), you do not need to write those loops anymore; your code will be more readable. For example, consider this Java code (where `strings` is a `List<String>`):

```
String found = null;
for (String string : strings) {
  if (string.startsWith("F")) {
    found = string;
    break;
  }
}
System.out.println(found);
```

The corresponding Xtend code using a lambda is as follows:

```
println(strings.findFirst([ s | s.startsWith("F")]))
```

In the Java version, you do not immediately understand what the code does. The Xtend version can almost be read as an English sentence (with some known mathematical notations): "find the first element s in the collection such that s starts with an F".

Xtend provides some additional syntactic sugar for lambdas to make code even more readable.

First of all, when a lambda is the last argument of a method invocation, it can be put outside the `(...)` parentheses (and if the invocation only requires one argument, the `()` can be omitted):

```
Collections::sort(list)[arg0, arg1 |
  arg0.compareToIgnoreCase(arg1)]
strings.findFirst[ s | s.startsWith("F") ]
```

Furthermore, the special symbol `it` we introduced earlier is also the default parameter name in a lambda expression; thus, if the lambda has only one parameter, you can avoid specifying it and instead use `it` as the implicit parameter:

```
strings.findFirst[ it.startsWith("F") ]
```

and since all the members of `it` are implicitly available without using `"."`, you can simply write the following:

```
strings.findFirst[startsWith("F")]
```

This is even more readable. If you need to define a lambda that takes no parameter at all, you need to make it explicit by defining an empty list of parameters before the `|` symbol. For example, the following code will issue a compilation error since the lambda (implicitly) expects one argument:

```
val l = [println("Hello")]
l.apply()
```

The correct version should be expressed as follows:

```
val l = [ | println("Hello")]
l.apply()
```

When implementing checks and generating code for a DSL, most of the time you will have to inspect models and traverse collections; all of the preceding features will help you a lot with this. If you are still not convinced, let us try an exercise: suppose you have the following list of `Person` (where `Person` is a class with string fields `firstname`, `surname`, and an integer field `age`):

```
personList = newArrayList(
  new Person("James", "Smith", 50),
  new Person("John", "Smith", 40),
  new Person("James", "Anderson", 40),
  new Person("John", "Anderson", 30),
  new Person("Paul", "Anderson", 30))
```

and we want to find the first three younger persons whose first name starts with J, and we want to print them as "surname, firstname" on the same line separated by "; ", thus, the resulting output should be (note: ; must be a separator):

```
Anderson, John; Smith, John; Anderson, James
```

Try to do that in Java; in Xtend (with lambdas and extension methods), it is as simple as follows:

```
val result = personList.filter[firstname.startsWith("J")].
  sortBy[age].
  take(3).
  map[surname + ", " + firstname].
  join("; ")
println(result)
```

Multi-line template expressions

Besides traversing models, when writing a code generator, most of the time you will write strings that represent the generated code; unfortunately, also for this task, Java is not ideal. In fact, in Java, you cannot write multi-line string literals.

This actually results in two main issues: if the string must contain a newline character, you have to use the special character \n; if, for readability, you want to break the string literal in several lines, you have to concatenate the string parts with +.

If you have to generate only a few lines, this might not be a big problem; however, a generator of a DSL usually needs to generate lots of lines.

For example, let us assume that you want to write a generator for generating some Java method definitions; you can write a Java class with methods that take care of generating specific parts, as shown in the following code:

```
public class JavaCodeGenerator {
  public String generateBody(String name, String code) {
    return "/* body of " + name + " */\n" + code;
  }

  public String generateMethod(String name, String code) {
    return "public void " + name + "() {" +
        "\t" + generateBody(name, code) +
        "}";
  }
}
```

You can then invoke it as follows:

```
JavaCodeGenerator generator = new JavaCodeGenerator();
System.out.println(generator.generateMethod("m",
  "System.out.println(\"Hello\"));\nreturn;"));
```

We can, however, spot some drawbacks of this approach:

- The shape of the final generated code is not immediately understandable (the variable parts break the constant parts)

- Code indentation is not trivial to handle

- Newline and tab characters are not easy to spot

- Some recurrent characters must be manually escaped (for example, the `""` quotations)
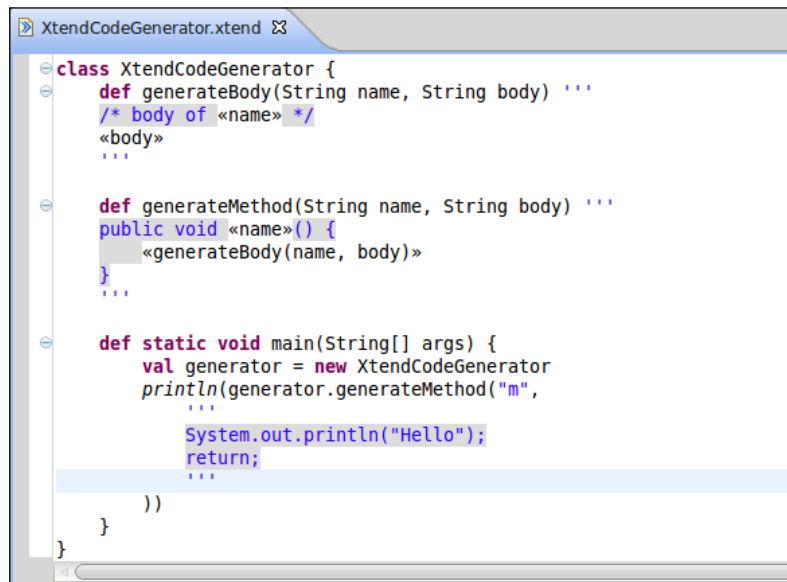
The result of running this generator is as follows:

```
public void m() {    /* body of m */
System.out.println("Hello"));
return;}
```

Although the generated code is a valid Java code, it is not nicely formatted (the Java compiler does not care about formatting, but it would be good to generate nicely formatted code, since the programmer might want to read it or debug it). This is due to the way we wrote the methods in the class `JavaCodeGenerator`: the code is buggy with this respect, but it is not easy to get this right when using Java strings.

Xtend provides multi-line template expressions to address all of the preceding issues (indeed, all strings in Xtend are multi-line). The corresponding code generator written in Xtend using multi-line template expressions is shown in the following screenshot:

```
XtendCodeGenerator.xtend ⊠
class XtendCodeGenerator {
    def generateBody(String name, String body) '''
    /* body of «name» */
    «body»
    '''

    def generateMethod(String name, String body) '''
    public void «name»() {
        «generateBody(name, body)»
    }
    '''

    def static void main(String[] args) {
        val generator = new XtendCodeGenerator
        println(generator.generateMethod("m",
            '''
            System.out.println("Hello");
            return;
            '''
        ))
    }
}
```

Before explaining the code, we must first mention that the final output is nicely formatted as it was meant to be:

```
public void m() {
  /* body of m */
  System.out.println("Hello");
  return;
}
```

Template expressions are defined using triple single quotes (this allows us to use double quotes directly without escaping them); they can span multiple lines, and a newline in the expression will correspond to a newline in the final output. Variable parts can be directly inserted in the expression using **guillemets** («», also known as **angle quotes** or **French quotation marks**). Note that between the guillemets, you can specify any expression, and you can even invoke methods. You can also use conditional expressions and loops (we will see an example later in this book; you can refer to the documentation for all the details).

> Curly brackets { } are optional for Xtend method bodies that only contain template expressions.

Another important feature of template expressions is that indentation is handled automatically and in a smart way. As you can see from the previous screenshot, the Xtend editor uses a specific syntax coloring strategy for multi-line template strings, in order to give you an idea of what the indentations will look like in the final output.

> To insert the guillemets in the Xtend Eclipse editor, you can use the keyboard shortcuts *Ctrl + Shift + <* and *Ctrl + Shift + >* for « and » respectively. On a Mac operating system, they are also available with *Alt + q* («) and *Alt + Q* (»). Alternatively, you can use content assist inside a template expression to insert a pair of them.

The drawback of guillemets is that you will have to have a consistent encoding, especially if you work in a team using different operating systems. You should always use **UTF-8** encoding for all the projects that use Xtend to make sure that the right encoding is stored in your project preferences (which is in turn saved on your versioning system, such as Git). You should right-click on the project and then select **Properties...**, and in the **Resource** property, set the encoding explicitly (refer to the following screenshot). You must set this property before writing any Xtend code (changing the encoding later will change all the guillemets characters, and you will have to fix them all by hand). Systems such as Windows use a default encoding that is not available in other systems, such as Linux, while **UTF-8** is available everywhere.
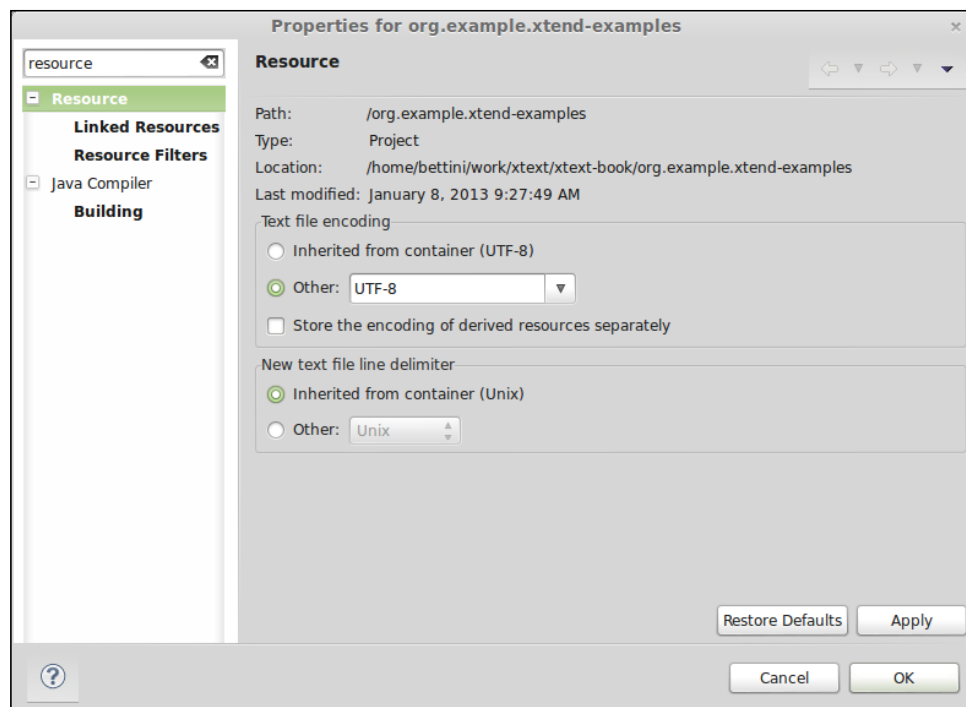
Additional operators

Besides standard operators, Xtend has additional operators that helps to keep code compact.

First of all, most standard operators are extended to lists with the expected meaning. For example, when executing the following code:

```
val l1 = newArrayList("a")
l1 += "b"
val l2 = newArrayList("c")
val l3 = l1 + l2
println(l3)
```

the string `[a, b, c]` will be printed.

Quite often, you will have to check whether an object is not null before invoking a method on it, otherwise you may want to return null (or simply perform no operation). As you will see in DSL development, this is quite a recurrent situation. Xtend provides the operator `?.`, which is the **null-safe** version of the standard selection operator (the dot `.`). Writing `o?.m` corresponds to `if (o != null) o.m`. This is particularly useful when you have cascade selections, for example, `o?.f?.m`.

The **Elvis** `?:` operator is another convenient operator for dealing with default values in case of null instances. It has the following semantics: `x ?: y` returns `x` if it is not null and `y` otherwise.

Combining the two operators allows you to set up default values easily, for example:

```
// equivalent to: if (o != null) o.toString else 'default'
result = o?.toString ?: 'default'
```

The **with** operator (or **double arrow** operator), `=>`, binds an object to the scope of a lambda expression in order to do something on it; the result of this operator is the object itself. Formally, the operator `=>` is a binary operator that takes an expression on the left-hand side and a lambda expression with a single parameter on the right-hand side: the operator executes the lambda expression with the left-hand side as the argument. The result is the left operand after applying the lambda expression.

For example, the code:

```
return eINSTANCE.createEntity => [ name = "MyEntity"]
```

is equivalent to:

```
val entity = eINSTANCE.createEntity
entity.name = "MyEntity"
return entity
```

This operator is extremely useful in combination with the implicit parameter `it` and the syntactic sugar for getters and setters to initialize a newly created object to be used in a further assignment without using temporary variables (again increasing code readability). As a demonstration, consider the Java code snippet we saw in *Chapter 2, Creating Your First Xtext Language,* that we used to build an `Entity` with an `Attribute` (with its type) that we will report here for convenience:

```java
Entity entity = eINSTANCE.createEntity();
entity.setName("MyEntity");
entity.setSuperType(superEntity);
Attribute attribute = eINSTANCE.createAttribute();
attribute.setName("myattribute");
AttributeType attributeType = eINSTANCE.createAttributeType();
attributeType.setArray(false);
attributeType.setDimension(10);
EntityType entityType = eINSTANCE.createEntityType();
entityType.setEntity(superEntity);
attributeType.setElementType(entityType);
attribute.setType(attributeType);
entity.getAttributes().add(attribute);
```

This requires many variables that are a huge distraction (are you able to get a quick idea of what the code does?). In Xtend, we can simply write:

```
eINSTANCE.createEntity => [
  name = "MyEntity"
  superType = superEntity
  attributes += eINSTANCE.createAttribute => [
    name = "myattribute"
    type = eINSTANCE.createAttributeType => [
      array = false
      dimension = 10
      elementType =  eINSTANCE.createEntityType => [
        entity = superEntity
      ]
    ]
  ]
]
```

Polymorphic method invocation

Method overloading resolution in Java (and by default in Xtend) is a static mechanism, meaning that the selection of the specific method takes place according to the static type of the arguments. When you deal with objects belonging to a class hierarchy, this mechanism soon shows its limitation: you will probably write methods that manipulate multiple polymorphic objects through references to their base classes, but since static overloading only uses the static type of those references, having multiple variants of those methods will not suffice. With **polymorphic method invocation** (also known as **multiple dispatch**, or **dynamic overloading**), the method selection takes place according to the runtime type of the arguments.

Xtend provides **Dispatch Methods** for polymorphic method invocation: upon invocation, overloaded methods marked as `dispatch` are selected according to the runtime type of the arguments.

Going back to our Entities DSL of *Chapter 2*, *Creating Your First Xtext Language*, `ElementType` is the base class of `BasicType` and `EntityType`, and `AttributeType` has a reference, `elementType`, to an `ElementType`; to have a string representation for such a reference, we can write two dispatch methods as in the following example:

```
def dispatch typeToString(BasicType type) {
  type.typeName
}
def dispatch typeToString(EntityType type) {
  type.entity.name
}
```

Now when we invoke `typeToString` on the reference `elementType`, the selection will use the runtime type of that reference:

```
def toString(AttributeType attributeType) {
  attributeType.elementType.typeToString
}
```

With this mechanism, you can get rid of all the ugly `instanceof` cascades (and explicit class casts) that have cluttered many Java programs.

Enhanced switch expressions

Xtend provides a more powerful version of Java `switch` statements. First of all, only the selected `case` is executed, in comparison to Java that falls through from one case to the next (thus, you do not have to insert an explicit break instruction to avoid subsequent case block execution); furthermore, a switch can be used with any object reference.

Xtend switch expressions allow you to write involved case expressions, as shown in the following example:

```
def String switchExample(Entity e, Entity specialEntity) {
  switch e {
    case e.name.length > 0 : "has a name"
    case e.superType != null : "has a super type"
    case specialEntity : "special entity"
    default: ""
  }
}
```

If the case expression is a `boolean` expression (like the first two cases in the preceding example), then the case matches if the case expression evaluates to `true`. If the case expression is not of type `boolean`, it is compared to the value of the main expression using the `equals` method (the third case in the preceding example). The expression after the colon of the matched case is then evaluated, and this evaluation is the result of the whole switch expression.

Another interesting feature of Xtend switch expressions is **type guards**. With this functionality, you can specify a type as the case condition and the case matches only if the switch value is an instance of that type (formally, if it conforms to that type). In particular, if the switch value is a variable, that variable is automatically casted to the matched type within the case body. This allows us to implement a cleaner version of the typical Java cascades of `instanceof` and explicit casts. Although we could use dispatch methods to achieve the same goal, switch expressions with type guards can be a valid and more compact alternative.

For example, the code in the previous section using dispatch methods can be rewritten as follows:

```
def toString(AttributeType attributeType) {
  val elementType = attributeType.elementType
  switch elementType {
    BasicType : // elementType is a BasicType here
      elementType.typeName
    EntityType: // elementType is an EntityType here
      elementType.entity.name
  }
}
```

Note how `entityType` is automatically casted to the matched type in the case body.

Depending on your programming scenario, you might want to choose between dispatch methods and type-based switch expressions; however, keep in mind that while dispatch methods can be overridden and extended (that is, in a derived class, you can provide an additional dispatch method for a combination of parameters that was not handled in the base class), switch expressions are inside a method, and thus they do not allow for the same extensibility features. Moreover, dispatch cases are automatically reordered with respect to type hierarchy (most concrete types first), while switch cases are evaluated in the specified order.

Debugging Xtend code

The Java code generated by Xtend is clean and easy to debug. However, it is also possible to debug Xtend code directly (instead of the generated Java), thanks to the complete integration of Xtend with the Eclipse JDT debugger.

This means that you can start debugging a Java application that at some point invokes Java code that has been generated by Xtend and, stepping through that, automatically brings you to debugging the original Xtend source. The next screenshot shows a debugging session of Xtend code. Note that all the JDT debugger views are available; furthermore, implicit variables such as it can be inspected in the **Variables** view:
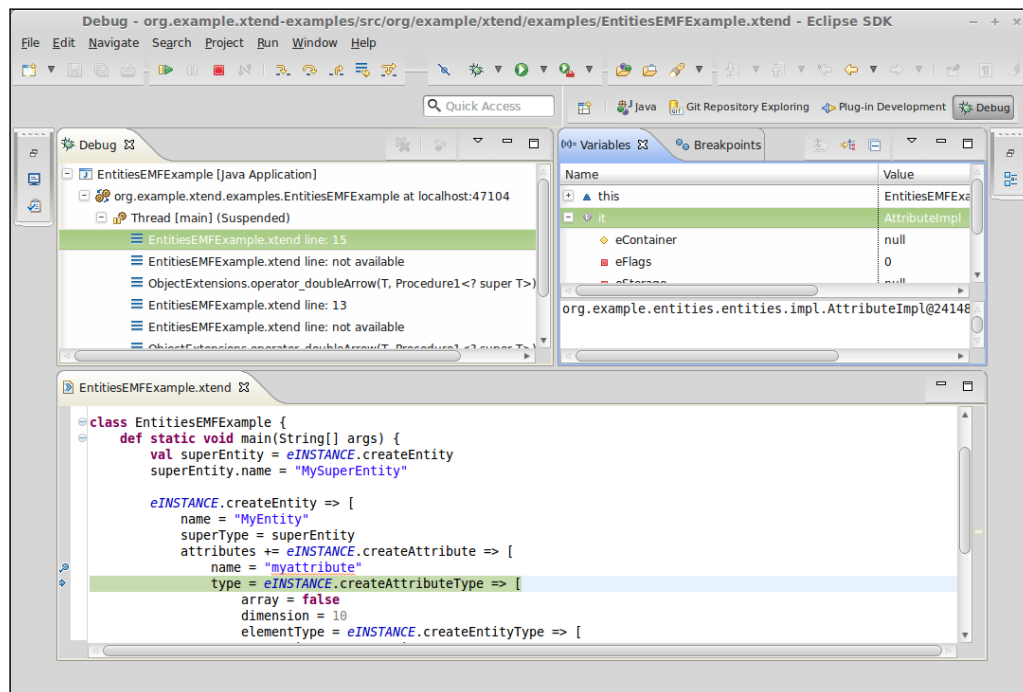
You can also debug an Xtend file (for example, containing a `main` method) directly, since all the Run and Debug configuration launches are available for Xtend files as well. Breakpoints can be inserted in an Xtend file by double-clicking on the breakpoint ruler in the editor (currently, the **Debug** context menu is not available for Xtend files).

If, for any reason, while debugging Xtend code you need to debug the generated Java code, you can do so by right-clicking on the **Debug** view on an element corresponding to an Xtend file line and selecting **Show Source** (refer the following screenshot).
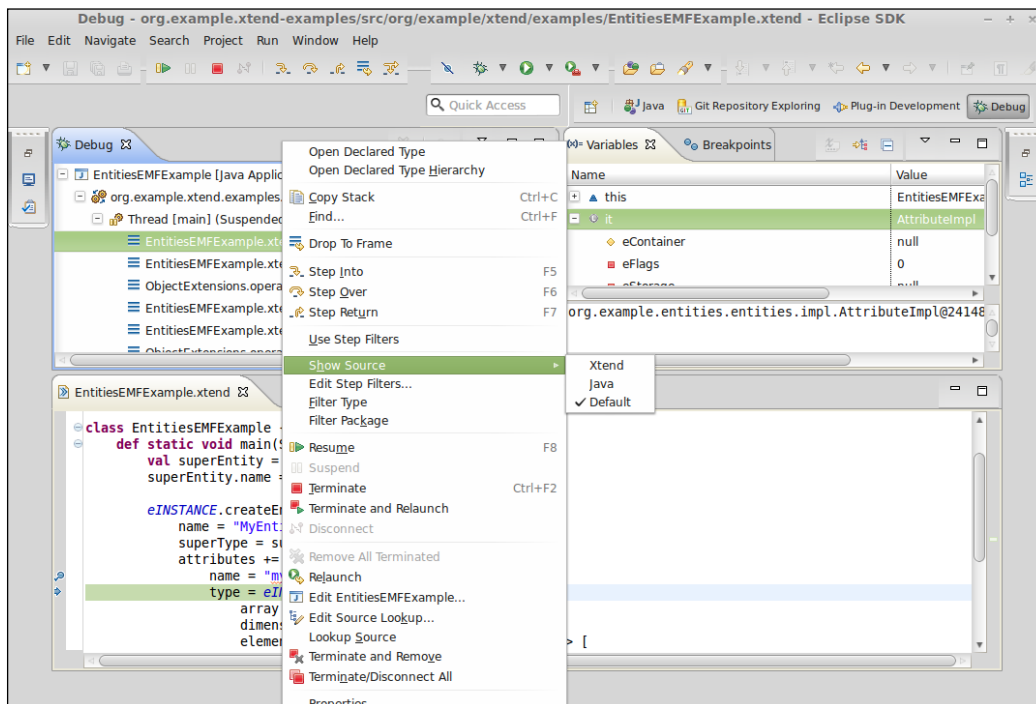
Summary

Xtend provides many features that allow you to write clean and more readable code. Since it is completely integrated with Java, all the Java libraries are accessible from within Xtend; moreover, the IDE tooling of Xtend itself is basically the same as the ones of JDT. For all of the aforementioned reasons, Xtext fosters the use of Xtend to develop all the aspects of a DSL implementation.

In the next chapter, we will show you how to implement constraint checks for a DSL using the EMF Validator mechanism using the Xtext enhanced API.

4

Validation

In this chapter we will introduce the concept of validation, and in particular, the Xtext mechanism to implement validation: the validator. With validation you can implement additional constraint checks of a DSL which cannot be done at parsing time. Xtext allows you to implement such constraint checks in an easy and declarative way; furthermore, you only need to communicate to Xtext the possible errors or warnings and it will take care of generating the error markers accordingly in the IDE. The validation will take place in the background while the user of the DSL is typing in the editor, so that an immediate feedback is provided. We will also show how to implement quickfixes corresponding to the errors and warnings generated during validation, so that we can help the user to solve problems due to validation errors.

Validation in Xtext

As we hinted in *Chapter 1*, *Implementing a DSL*, parsing a program is only the first stage in a programming language implementation. In particular, the overall correctness of a program cannot always be determined during parsing. Trying to embed additional constraint checks in the grammar specification could either make such specification more complex or it could be simply impossible as some additional static analysis can be performed only when other program parts are already parsed.

Actually, the best practice is to do as little as possible in the grammar and as much as possible in validation (we will use this practice in *Chapter 9*, *Type Checking* and *Chapter 10*, *Scoping*). This is because it is possible to provide far better error messages and to more precisely detect problems that are eligible for quickfixes.

The mechanism of validation will be used extensively in all example DSLs of this book. Typically, even for small DSLs, a validator has to be implemented to perform additional constraint checks.

In Xtext, these constraints checks are implemented using a validator, which is a concept inherited from the corresponding EMF API (see Steinberg et al., 2008). In EMF, you can implement a validator which performs constraint checks on the elements of an EMF model. Since Xtext uses EMF for representing the AST of a parsed program, the mechanism of the validator naturally extends to an Xtext DSL. In particular, Xtext enhances the EMF API for validation, by providing a declarative way to specify rules for constraints of your DSL. Moreover, Xtext comes with default validators, some of which are enabled by default, to perform checks which are common to many DSLs (for example, cross-reference checks). Your custom validator can be composed with the default ones of Xtext.

Default validators

Let us go back to the Entities DSL of *Chapter 2*, *Creating your First Xtext Language*. Since we expressed cross-references in our Entities grammar, we can see the Xtext linker/validator in action in the generated editor. If we enter an incorrect reference (for example, the name of a super entity that does not exist), we get the error "Couldn't resolve reference to...". This check on cross-references is performed by one of the default validators provided by Xtext (cross-reference resolution is the main subject of *Chapter 10*, *Scoping*).

Another standard validator provided by Xtext is the one that checks whether the names are unique within your program. This check validates that names are unique per element type; for example, you can have an attribute named the same as an entity, but not two entities with the same name.

A unique name validator is not enabled by default, but it can be turned on by modifying the MWE2 file as shown in the following code snippet. In our example, it is `GenerateEntities.mwe2`; in particular, we need to uncomment the `composedCheck` specification which concerns `NamesAreUniqueValidator`:

```
fragment = validation.ValidatorFragment auto-inject {
  composedCheck =
    "org.eclipse.xtext.validation.NamesAreUniqueValidator"
}
```

After that, of course, you need to run the MWE2 workflow.

If you now try to declare two entities with the same name in the Entities DSL editor, you will get an error as shown in the following screenshot:
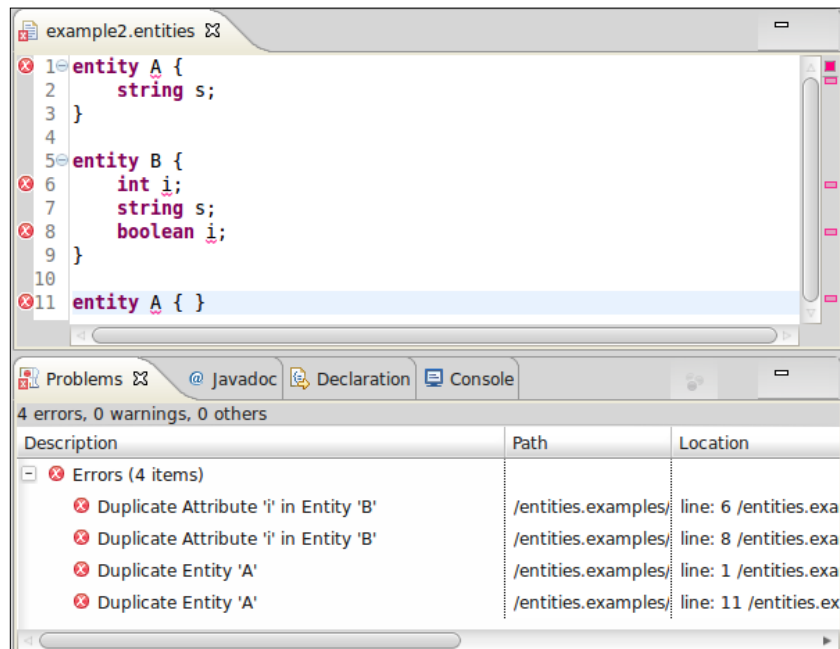
It is interesting to note that, besides the element type, this validator also takes into consideration the containment relations, for example, two attributes declared in two different entities are allowed to have the same name (as you can see from the preceding screenshot, both A and B have the attribute s, and this is allowed).
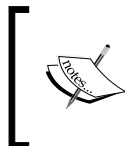
> Technically, everything that is referenceable via name is named in a namespace, implied by the containment relation. This leads to qualified names, which will be explained in *Chapter 10, Scoping*.

The default behavior of this validator should suit most DSLs. If your DSL needs to have more rigid constraints about names, or in general about duplicate elements, you will have to implement a customized `NamesAreUniqueValidator` class or simply disable `NamesAreUniqueValidator` and implement these name checks in your own validator (an example of a custom duplicate name check is shown in *Chapter 9, Type Checking*).

Custom validators

While the default validators can perform some common validation tasks, most of the checks for your DSL will have to be implemented by you, according to the semantics you want your DSL to have.

These additional checks can be implemented using the Xtend class that Xtext has generated for you in the `validation` subpackage in the `src` folder of the runtime plug-in project. In our example, this class is called `EntitiesValidator`. Remember that since this class is in the `src` folder, it will not be overwritten by future MWE2 workflow executions. Xtext performs validation by invoking each method annotated with `@Check`, passing all instances having a compatible runtime type to each such method. The name of the method is not important, but the type of the single parameter is important. You can define as many annotated methods as you want for the same type; Xtext will invoke them all. Inside such methods you can implement the semantic checks for that element. If a semantic check fails, you can call the `error` method, which will be explained shortly.

For example, we want to make sure that there is no cycle in the hierarchy of an entity; thus, we write the following annotated method in our validator:

```
class EntitiesValidator extends AbstractEntitiesValidator {

  @Check
  def checkNoCycleInEntityHierarchy(Entity entity) {
    if (entity.superType == null)
      return // nothing to check

    val visitedEntities = <Entity>newHashSet()
    visitedEntities.add(entity)
    var current = entity.superType
    while (current != null) {
      if (visitedEntities.contains(current)) {
        error("cycle in hierarchy of entity '"+current.name+"'",
          EntitiesPackage::eINSTANCE.entity_SuperType)
        return
      }
      visitedEntities.add(current)
      current = current.superType
    }
  }
}
```

In the preceding method we traverse the hierarchy of an entity (of course, if an entity has no `superType`, there is nothing to check) by recording all the entities we are visiting. If during this visit we find an entity that we have already visited it means that the hierarchy contains a cycle and we issue an error. It is crucial to leave the while loop in that case, otherwise the loop will never end (after all, we found a cycle and we would traverse the hierarchy endlessly).

The method `error` (we refer to Xtext documentation for further details) has many overloaded versions. In this example, we use the version that requires the following:

- A message for the error (and it is up to you to provide meaningful information).
- The EMF feature of the examined `EObject` which the error should be reported against, that is, which should be marked with error. In this case, the feature containing the error is the `superType` feature.

Access to classes and features are obtained from the `EPackage` class that is generated for our DSL's metamodel (in our example ,`EntitiesPackage`). Using this `EPackage`, EMF features can be obtained in two ways:

- Using the static instance of the package and then the method corresponding to the feature (as we did in the preceding code):

  `EntitiesPackage::`*`eINSTANCE`*`.entity_SuperType.`

- Using the static fields of the inner interface `Literals`:

  `EntitiesPackage$Literals::`*`ENTITY__SUPER_TYPE`*`.`

In both cases, in your Xtend programs, you can rely on the content assist to select the feature easily.

We can now try the above validation check in the Entities DSL editor by defining entities which contain a cycle in the hierarchy, as shown in the following screenshot:
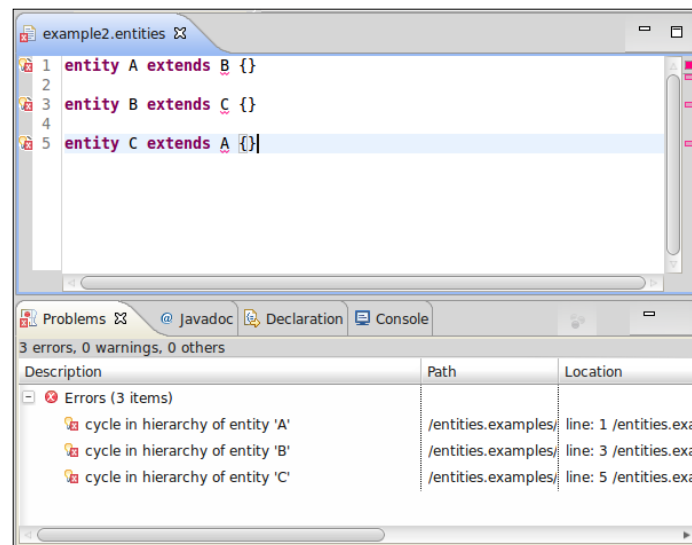
You can see that the highlighted element in the editor is the entity name after the keyword extends, since that corresponds to the feature superType.

The three error markers also show that Xtext calls our @Check annotated method for all the elements of type Entity in the program.

Calling the error method with the appropriate information will let Xtext manage the markers for Xtext based resources (clearing them before reparsing, keeping track of dirty versus saved state, and so on). Markers will appear wherever they are supported in the IDE: in the right and left editor ruler, in the Problems view, and in the package explorer.

If you want to issue warnings instead of errors (which are considered to mean that the model is invalid), simply call the warning method that has the same signature as the error method. For example, in our Entities DSL, we follow a standard convention about names: the name of an entity should start with a capital letter, while the name of an attribute should be lowercase. If the user does not follow this convention, we issue a warning; the program is considered valid anyway. To implement this, we write the following methods in the EntitiesValidator class (note the use of imported static methods as extension methods from the class Character):

```
import static extension java.lang.Character.*

class EntitiesValidator extends AbstractEntitiesValidator {

  ...
  @Check
  def checkEntityNameStartsWithCapital(Entity entity) {
    if (entity.name.charAt(0).lowerCase)
      warning("Entity name should start with a capital",
        EntitiesPackage::eINSTANCE.entity_Name)
  }

  @Check
  def checkAttributeNameStartsWithLowercase(Attribute attr) {
    if (attr.name.charAt(0).upperCase)
      warning("Attribute name should start with a lowercase",
        EntitiesPackage::eINSTANCE.attribute_Name)
  }
}
```

The following screenshot shows how warning markers are created instead of error markers:
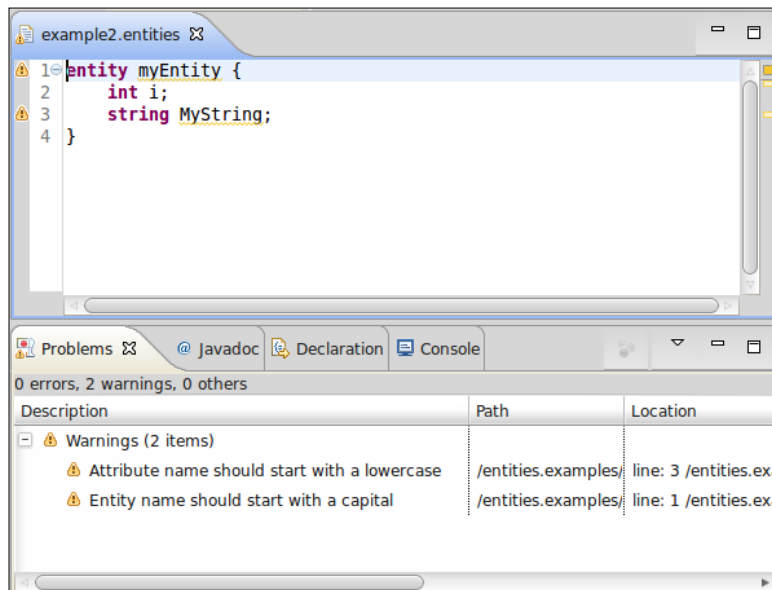
> In Xtext 2.4 an "info" severity level has been added (the method to call is info). In this case an information marker is shown only in the editor's ruler, while the corresponding file is not marked.

This is just an example of a simple validator implementation; in the rest of the book, we will see many other implementations that perform more complex constraint checks (among which, type checking, as shown in *Chapter 8*, *An Expression Language* and in *Chapter 9*, *Type Checking*).

Quickfixes

As we said in *Chapter 1*, *Implementing a DSL*, a quickfix is a proposal to solve a problem in a program. Quickfixes are typically implemented by a context menu available from the error marker, and they are available both in the editor ruler and in the Problems view.

> Since quickfixes are tightly connected to validation, we describe them in this chapter. Moreover, they allow us to get familiar with the manipulation of the EMF model representing the AST of a program.

In our Entities DSL we can provide a quickfix for each warning and error issued by our validator; moreover, as we will see later, we can also provide quickfixes for errors issued by Xtext default validators.

Xtext provides an easy mechanism to implement a quickfix connected to an error or warning issued by a validator. The Xtext generator generates an Xtend stub class for quickfixes into the UI plug-in project; in our Entities DSL example, this class is `org.example.entities.ui.quickfix.EntitiesQuickfixProvider`.

A quickfix is triggered by an issue code associated with an error or warning marker. An issue code is simply a string that uniquely identifies the issue. Thus, when invoking the `error` or `warning` method, we must provide an additional argument which represents the issue code. In the validator, this is typically done by defining a public `String` constant, whose value is prefixed with the package name of the DSL and ends with a sensible name for the issue. It might also make sense to pass additional issue data that can be reused by the quickfix provider to show a more meaningful description of the quickfix and to actually fix the program. It is worth noting that issue data is very useful when validation needs to compute something that is costly; the quickfix may then avoid having to compute it again. Thus, we use another version of the method `error` and `warning` which takes four arguments, and we modify our validator as follows (only the modified parts are shown):

```
class EntitiesValidator extends AbstractEntitiesValidator {

  public static val HIERARCHY_CYCLE =
  "org.example.entities.HierarchyCycle";

  public static val INVALID_ENTITY_NAME =
  "org.example.entities.InvalidEntityName";

  public static val INVALID_ATTRIBUTE_NAME =
  "org.example.entities.InvalidAttributeName";

  @Check
  def checkNoCycleInEntityHierarchy(Entity entity) {
    ...
    error("cycle in hierarchy of entity '"+current.name+"'",
        EntitiesPackage::eINSTANCE.entity_SuperType,
        HIERARCHY_CYCLE,
        current.superType.name)
    ...
  }

  @Check
  def checkEntityNameStartsWithCapital(Entity entity) {
    if (entity.name.charAt(0).lowerCase)
```

```
        warning("Entity name should start with a capital letter",
          EntitiesPackage::eINSTANCE.entity_Name,
          INVALID_ENTITY_NAME,
          entity.name)
    }

    @Check
    def checkAttributeNameStartsWithLowercase(Attribute attr) {
      if (attr.name.charAt(0).upperCase)
        warning("Attribute name should start with a lowercase",
          EntitiesPackage::eINSTANCE.attribute_Name,
          INVALID_ATTRIBUTE_NAME,
          attr.name)
    }
}
```

The issue constant is passed as the third argument to the methods `error` and `warning`. Issue data is optional and you can pass a variable number of issue data arguments. To implement a quickfix, we define a method in `EntitiesQuickfixProvider` annotated with `@Fix` and a reference to the issue code this quickfix applies to. The name of the method is not important, but the parameter types are fixed.

For example, for the warning concerning the first letter of an entity name, which must be capital, we implement a quickfix which automatically capitalizes the first letter of that entity:

```
class EntitiesQuickfixProvider extends DefaultQuickfixProvider {

  @Fix(EntitiesValidator::INVALID_ENTITY_NAME)
  def void capitalizeEntityNameFirstLetter(Issue issue,
                             IssueResolutionAcceptor acceptor) {
    acceptor.accept(issue,
      "Capitalize first letter", // label
      "Capitalize first letter of '"
        + issue.data.get(0) + "'", // description
      "Entity.gif", // icon
      [
        context |
        val xtextDocument = context.xtextDocument
        val firstLetter = xtextDocument.get(issue.offset, 1);
        xtextDocument.replace(issue.offset, 1,
                             firstLetter.toFirstUpper);
      ]
    );
  }
```

Let us analyze what this code does. The first parameter of a quickfix provider method is the `Issue` object that represents the error information; this is built internally by Xtext using the information passed to `error` or `warning` in your validator. The second parameter is an **accepto**r. Acceptor is a concept, and you will see different types of acceptors used in many places in Xtext; you usually only have to invoke the method `accept` on an acceptor, passing some arguments.

The first three arguments passed to the method `accept` of the acceptor are the label (shown in the quickfix pop up for this fix), a description (which should show what the effect of selecting this quick fix would mean, or something that makes the user confident it is a fix they want to apply), and an icon (if you do not want an icon, you can pass an empty string; how to use custom icons in your DSL UI will be explained in *Chapter 6*, *Customizations*). Note that for the description, we use the first element of the issue data (an array) since we know that in the validator we passed the name of the supertype as the single issue data (when implementing the quickfix you must be consistent with the information passed by the validator).

The fourth argument is the most interesting one, since it is actually the modification code performed by the quickfix when selected by the user. In Java, the fourth argument would be an inner class, but in Xtend, we can pass a lambda. Quickfixes can perform the correction based on the source text (textual modification), or on the model (semantic modification). These are explained in the next two sections.

Textual modification

You can specify a lambda which takes a single parameter of the type `IModificationContext`. Due to the powerful type inference mechanisms of Xtend, it is enough to just specify the name of the parameter (and Xtend will infer its type). We could have also specified no parameter at all, since by default, lambdas have automatically one parameter named `it`. In our example, the name of the parameter was explicitly stated for clarity.

In the preceding code, we use the `IDocument` argument, which is passed in the given modification context, to get access to the text we want to modify in our quickfix. We have been given the offset and length of where the error/warning is marked in the `Issue` object. We can now use the document methods `get(offset, length)` and `replace()` to perform the capitalization of the first letter.

This quickfix is shown in the following screenshot.
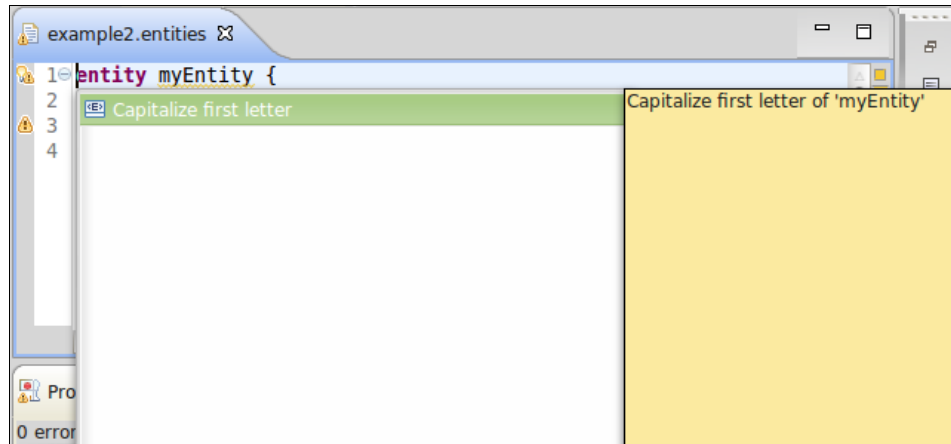
Using this strategy for implementing quickfixes has the drawback that we need to deal with the actual text of the editor.

Model modification

The alternative strategy relies on the fact that the program is also available in memory as an EMF model: if we modify the model, the Xtext editor will automatically update its contents. For this reason, we can specify a lambda which takes two parameters: the EMF element (that is, the `EObject`) that contains the error, and the modification context. The `EObject` element is the one the warning was reported against.

For instance, to uncapitalize the first letter of an attribute, we can write the quickfix using the following strategy:

```
@Fix(EntitiesValidator::INVALID_ATTRIBUTE_NAME)
def void uncapitalizeAttributeNameFirstLetter(Issue issue,
                             IssueResolutionAcceptor acceptor) {
  acceptor.accept(issue,
    "Uncapitalize first letter", // label
    "Uncapitalize first letter of '"
       + issue.data.get(0) + "'", // description
    "Attribute.gif", // icon
    [
      element, context |
      (element as Attribute).name = issue.data.get(0).toFirstLower
    ]
  );
}
```

In this case, the element is the `Attribute` object against which the warning was reported. Therefore, we simply assign the fixed name to the name of the attribute. Note that with this strategy, we only manipulate the EMF model, without having to deal with the contents of the editor: Xtext will then take care of updating the editor's contents.

The ability to directly modify the EMF model of the program makes more complex quickfixes easier to implement. For example, if we want to implement the quickfix to remove the supertype of the entity which contains a cycle in the hierarchy, we just need to set the `superType` feature to null, as shown in the following code snippet (you can see the quickfix in the following screenshot):

```
@Fix(EntitiesValidator::HIERARCHY_CYCLE)
def void removeSuperType(Issue issue,
                         IssueResolutionAcceptor acceptor) {
  acceptor.accept(issue,
    "Remove supertype",
    '''Remove supertype '«issue.data.get(0)»' ''',
    "delete_obj.gif",
    [ element, context |
      (element as Entity).superType = null;
    ]
  );
}
```
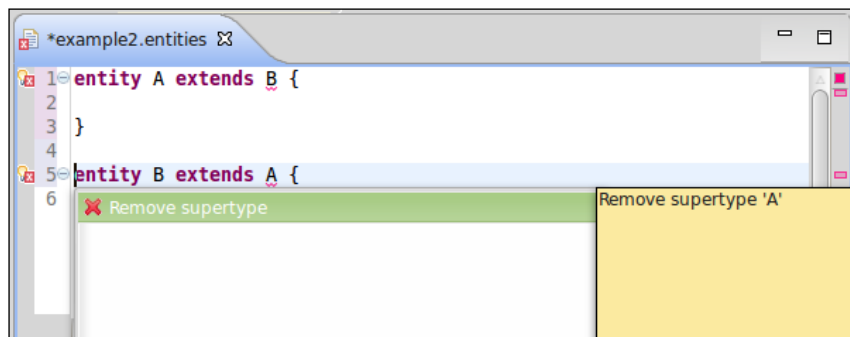
Note that the semantic change results in there not being any supertype in the model element and thus the source text `extends` is also removed. Implementing the same quickfix by modifying the text of the program would require more effort and would be more error prone. On the other hand, textual modifications allow fixing things that are not present in the semantic model. We could, for example, allow some punctuation to be optional in the grammar in order to provide better error messages such as "missing comma" as opposed to "syntax error". Such lenient behavior makes it possible for the parser to produce a model, and there is something semantic to base validation as opposed to a syntax error where we get no model at all. Also, semantic changes always include formatting, and this may have other unwanted side effects (we will deal with formatting in *Chapter 6*, *Customizations*).

Quickfixes for default validators

As mentioned before, we can also provide quickfixes for errors issued by default Xtext validators. You might have noted that if you refer to a missing entity in the Entities DSL editor, Xtext already proposes some quickfixes: if there are other entities in the source file, it proposes to change the reference to one of the existing entities. We can provide an additional quickfix that proposes to automatically add the missing entity. In order to do this, we must define a method in our quickfix provider for the issue `org.eclipse.xtext.diagnostics.Diagnostic.LINKING_DIAGNOSTIC`. We want to add the missing entity to the current model; to make things more interesting, the quickfix should add the missing entity after the entity which refers to the missing entity. For example, consider the following source file:

```
entity MyFirstEntity {
  FooBar s;
  int[] a;
}

entity MyOtherEntity {
}
```

The referred `FooBar` entity in the attribute definition is not defined in the program, and we would like to add it after the definition of `MyFirstEntity` (and before `MyOtherEntity`), since that is the entity which contains the attribute definition referring to the missing entity.

Let's first present the code for this quickfix (we will use Xtend features for getters, setters, template strings, and the with operator to make our logic more compact):

```
import static extension org.eclipse.xtext.EcoreUtil2.*

@Fix(Diagnostic::LINKING_DIAGNOSTIC)
```

```
    def void createMissingEntity(Issue issue,
                                 IssueResolutionAcceptor acceptor) {
      acceptor.accept(issue,
        "Create missing entity",
        "Create missing entity",
        "Entity.gif",
        [ element, context |
          val currentEntity =
              element.getContainerOfType(typeof(Entity))
          val model = currentEntity.eContainer as Model
          model.entities.add(
              model.entities.indexOf(currentEntity)+1,
              EntitiesFactory::eINSTANCE.createEntity() => [
                  name = context.xtextDocument.get(issue.offset,
                                                   issue.length)
              ]
          )
        ]
      );
    }
```

Consider that the `EObject` element passed to the lambda is the program element which refers to the missing entity, thus, it is not necessarily an `Entity` (for instance, if the missing entity is in an attribute's type specification, as in the preceding Entities program snippet, then the `EObject` element is an `AttributeType`). To get the containing entity, we could walk up the containment relation of the EMF model till we get to an `Entity` element. Alternatively, we can use one of the many static utility methods, here imported as `extension` methods, provided by Xtext in the class `EcoreUtil2` (this complements the standard `EcoreUtil` class of EMF). In particular, we use `getContainerOfType`, which does this walking up in the containment relation for us, until it finds an element of the specified type. For retrieving the root `Model` element we can simply cast the container of the found entity (since in our Entities DSL, an `Entity` can only be contained in a `Model`). Then, we insert the newly created entity in the desired position (that is, right after the position of the current entity).

> Spend some time to take a look at the classes `EcoreUtil` and `EcoreUtil2`, since they provide many useful methods you will need when dealing with an EMF model.

To create the missing entity, we must know its name. For this issue (which is not generated by our own validator), the issue data does not contain any information about the missing element's name. However, the issue offset tells us in which position in the document the missing element's name is referred. Thus, the name of the missing element can be retrieved using this offset (the length is also contained in the issue) from the editor's document.

You can now check what this quickfix does (see the following screenshot; this also shows the default quickfixes provided by Xtext which propose to change the name of the referred entity to one available in the current source).
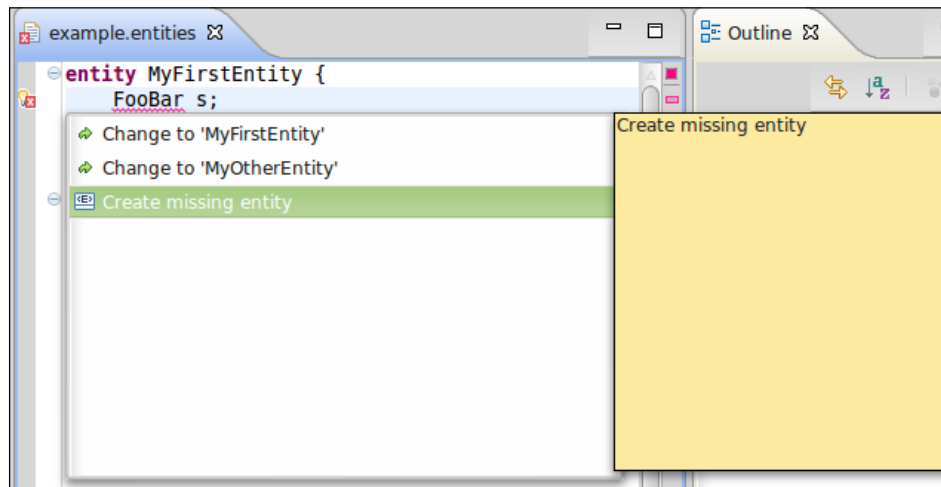
Summary

In this chapter you learned how to implement constraint checks, using the Xtext validator mechanism based on `@Check` annotated methods. Just by implementing a custom validator and calling the method `error` or `warning` with the appropriate information, Xtext produces error and warning markers that result in marking the text regions as well as showing the markers in the various views in Eclipse.

We also showed how to implement quickfixes. Since Xtext automatically synchronizes the DSL editor's contents with the EMF model of the AST, we can simply modify such model without dealing with the textual representation of the program.

In the next chapter we will write a code generator for the Entities DSL implemented in Xtend, relying on its advanced features for code generation: starting from a program written in our Entities DSL, we will generate the corresponding Java code. You will see that Xtext automatically integrates your code generator into the building infrastructure of Eclipse.

5
Code Generation

In this chapter you will learn how to write a code generator for your Xtext DSL using the Xtend programming language. Using the Entities DSL that we developed in the previous chapters, we will write a code generator which, for each entity declaration, will generate the corresponding Java class. We will also see how the code generator is automatically integrated by Xtext into the Eclipse builder infrastructure. Finally, the DSL implementation can be exported as a Java standalone command-line compiler.

Introduction to code generation

After a program written in your DSL has been parsed and validated, you might want to do something with the parsed EMF model, that is, the AST of that program. Typically, you may want to generate code in another language (for example, Java code), a configuration file, XML, a text file, and so on. In all of these cases, you will need to write a code generator.

Since the parsed program is an EMF model, you can indeed use any EMF framework which somehow deals with code generation. Of course, you might also use plain Java to generate code, after all, as we saw in the previous chapters, you have all the Java APIs to access the EMF model.

However, in this book, we will use Xtend (introduced in *Chapter 3*, *The Xtend Programming Language*) to write code generators, since it is very well suited for the task. Moreover, the task of writing a code generator is greatly simplified by the fact that Xtext automatically integrates your code generator into the Eclipse build infrastructure. All we have to deal with is producing the desired output (for example, Java source, XML, and so on).

Writing a code generator in Xtend

A generator stub is automatically created by Xtext. In our example the stub
is created in the package `org.example.entities.generator`:

```
class EntitiesGenerator implements IGenerator {
  override void doGenerate(Resource res, IFileSystemAccess fsa) {
    // TODO implement me
  }
}
```

Before writing the actual code, let us recall that Xtext is a framework, thus, the
overall flow of control is dictated by the framework, not by the programmer
(this is also known as the **Hollywood Principle**: "Don't call us, we'll call you").
This means that you do not have to run the generator; your DSL Xtext editor is
already integrated in the automatic building infrastructure of Eclipse, and the
generator will be automatically called when a source written in your DSL changes
(indeed, it will be called also if one of its dependencies changes, as we will see in
later chapters). Note that the method you have to implement just accepts an EMF
`Resource`, which contains the EMF model representation of the program. This means
that if this generation method is invoked, the corresponding source program has
already been parsed and validated: it will only be called when the source program
does not have any validation errors. You do not even have to worry about the
physical base path location where your code will be generated: that is hidden in the
passed `IFileSystemAccess` file. You only need to specify the relative path where
the generated file will be created and its contents as a string (actually, it can be any
`java.lang.CharSequence`).

First of all, we need to decide what we want to generate from our DSL programs.
For the Entities DSL it might make sense to generate a Java class for each `Entity`.
Thus, we need to retrieve all the `Entity` objects in the passed `Resource`. Here is how
to do this with one line of Xtend code:

```
resource.allContents.toIterable.filter(typeof(Entity))
```

Then, we iterate over these entities and generate a Java file for each of them using the
entity's name for the file name. Since we do not have explicit packages in our DSL,
we choose to generate all the Java classes in the package `entities`. Thus, the Java file
path will be:

```
"entities/" + entity.name + ".java"
```

Remember that we do not have to care about the base path (it will be configured into the passed `IFileSystemAccess` file). Summarizing we get:

```
override void doGenerate(Resource res, IFileSystemAccess fsa) {
  for (e : res.allContents.toIterable.filter(typeof(Entity))){
    fsa.generateFile(
      "entities/" + e.name + ".java",
      e.compile)
  }
}
```

Now we have to implement the `compile` method, which must return a string with the contents that will be stored in the generated file. We use Xtend multi-line template expressions to implement such a method; the method is illustrated in the following screenshot:

```
def compile(Entity entity) {
	'''
	package entities;

	public class «entity.name» «IF entity.superType != null»extends «entity.superType.name» «ENDIF»{
		«FOR attribute : entity.attributes»
		private «attribute.type.compile» «attribute.name»;
		«ENDFOR»

		«FOR attribute : entity.attributes»
		public «attribute.type.compile» get«attribute.name.toFirstUpper»() {
			return «attribute.name»;
		}

		public void set«attribute.name.toFirstUpper»(«attribute.type.compile» _arg) {
			this.«attribute.name» = _arg;
		}

		«ENDFOR»
	}
	'''
}
```

Note that the Xtend editor also shows the tab indentations of the final resulting string. Xtend smartly ignores indentations of the loop constructs, which are there only to make Xtend code more readable; these indentations and newline characters will not be a part of the resulting string.

We basically write a template for a Java class using the information stored in the `Entity` object. We generate the `extends` part only if the entity has a supertype. This is achieved using the `IF` conditional inside the template. Then, we iterate over the entity's attributes twice using the template `FOR` loop construct; the first time to generate Java fields and the second time to generate getter and setter methods. Doing that in two separate iterations allows us to generate all the fields at the beginning of the Java class. We use the `toFirstUpper` extension method to correctly generate the names of the getter and setter methods.
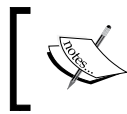
> Notice that `IF` and `FOR` (with capital letters) are used to specify conditions and loops, respectively, within a template expression.

We delegate the compilation of attribute types to another method:

```
def compile(AttributeType attributeType) {
  attributeType.elementType.typeToString +
    if (attributeType.array) "[]"
    else ""
}

def dispatch typeToString(BasicType type) {
  if (type.typeName == "string") "String"
  else type.typeName
}

def dispatch typeToString(EntityType type) {
  type.entity.name
}
```

Observe that the `BasicType` literals of our DSL already correspond to Java basic types; the only exception is "string", which in Java corresponds to "String". To keep the example simple, we do not consider the `length` feature of `AttributeType` during the code generation.

Integration with the Eclipse build mechanism

It is time to see our generator in action: launch Eclipse, create a Java project in the workspace, and, in the `src` folder, create a new `.entities` file (remember to accept to add the Xtext nature to the project, otherwise the generator will not run). Continue by adding one or more entities with some attributes. Notice that a `src-gen` folder is automatically created as soon as you save the file. At this point, you should also add this generated folder to the projects source folders by going to **Build Path | Use as Source Folder**. Exploring the content of the `src-gen` folder you will find a generated Java class for each entity in your `.entities` file. You can see an example in the following screenshot.

> Observe that a code generator just creates text. Other components have to make sense of that, for example, a Java compiler. That is why we need to add the `src-gen` folder to the project source folders: this way, the Eclipse Java compiler automatically compiles the generated Java sources.
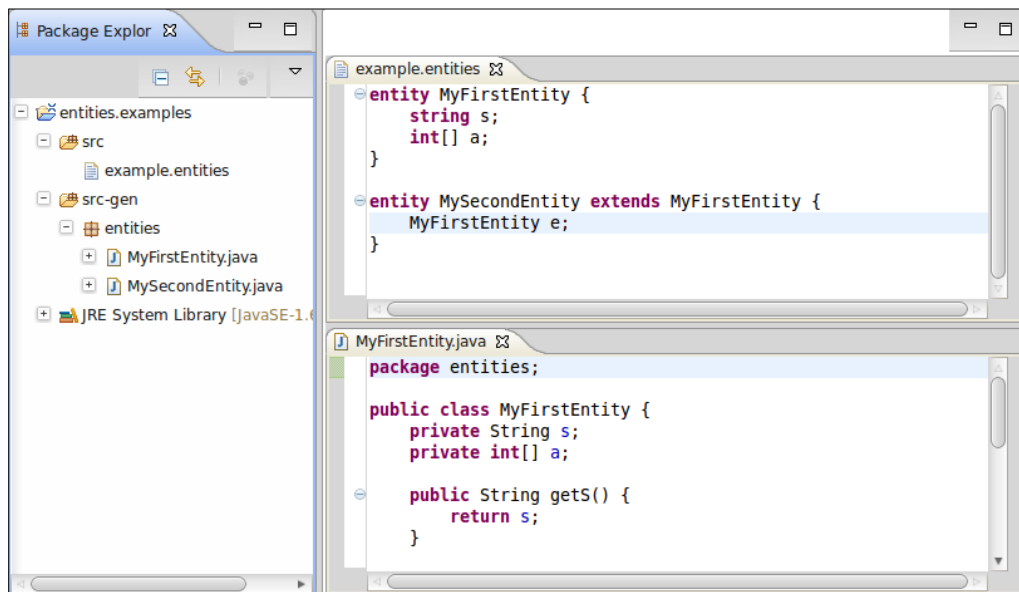
Make some changes and observe that the Java files are regenerated as you save the changes. Remove an entity and observe that the corresponding Java file is removed. Your DSL generator is completely integrated in the Eclipse build system.

For instance, in the following screenshot, we can see how the generated Java file is changed after removing a field from `MyFirstEntity` and how the previously generated Java file for `MySecondEntity` is automatically deleted after we removed the corresponding entity definition in the input file:
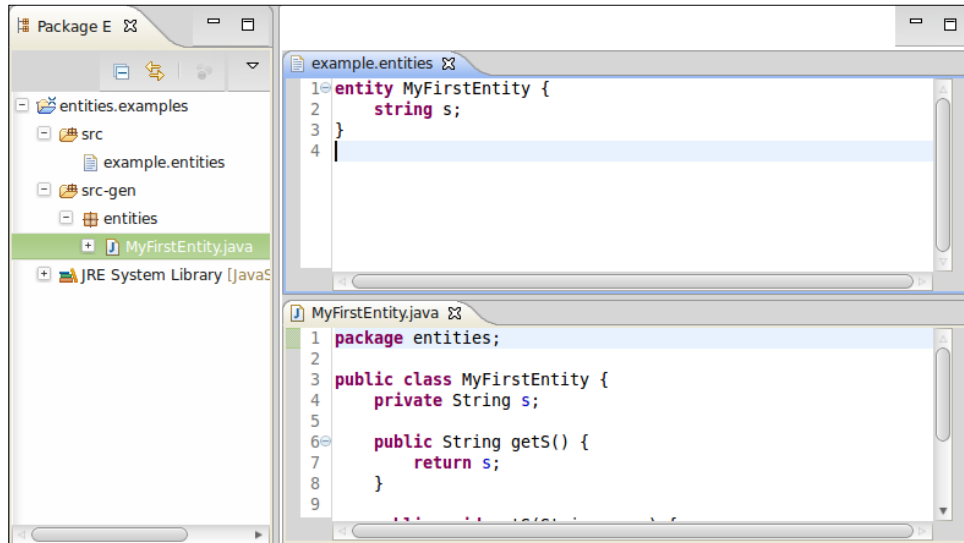
As we hinted before, the generator is automatically invoked by Xtext when the input file changes, provided that the file contains no validation error. Xtext also automatically keeps track of the association between the generated files and the original input file (as in this example, from an input file we can generate several output files). When a DSL source file is found to be invalid (that is, having validation errors) everything that was generated from that file is automatically removed, independent of the input program's element that contains the error.

For instance, in the following screenshot, we show what happens if we modify the `example.entities` file introducing an error: even if the error does not concern `MyFirstEntity`, its corresponding generated Java file is still deleted:
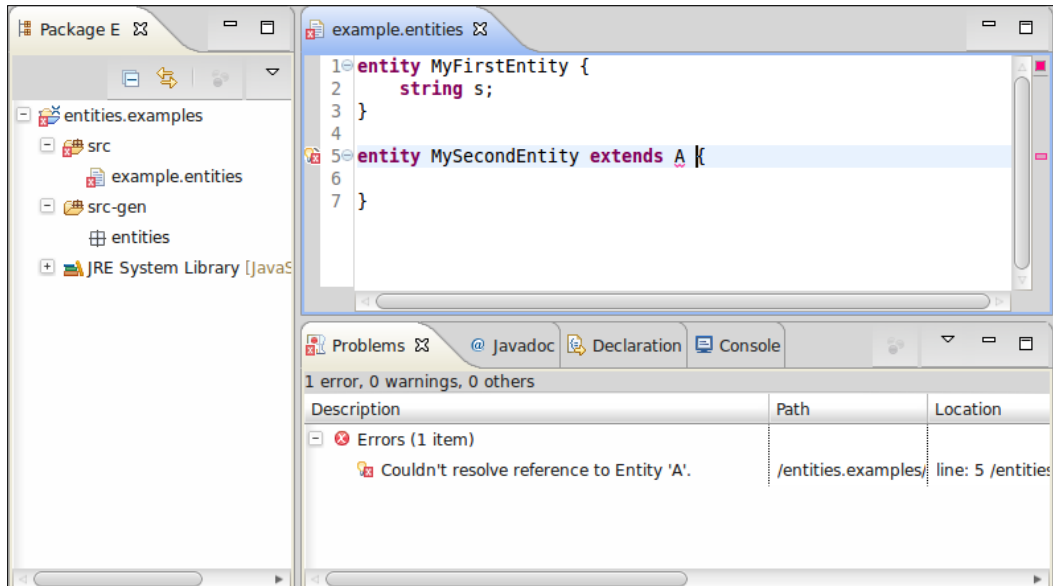
> The integration with the Eclipse build infrastructure is customizable, but the default behavior, including the automatic removal of generated files corresponding to an invalid source file, should fit most cases. In fact, code generators tend to be written for complete and valid models and often throw exceptions for incomplete models or produce complete garbage.

When running the MWE2 workflow, Xtext also creates preference pages for your DSL. One of these preference pages concerns code generation. In the new Eclipse instance you can check what Xtext created (go to **Window | Preferences**); you will see that there is a dedicated section for the Entities DSL with typical configurations (for example, syntax highlighting colors and fonts and code generation preferences), see the following screenshot:
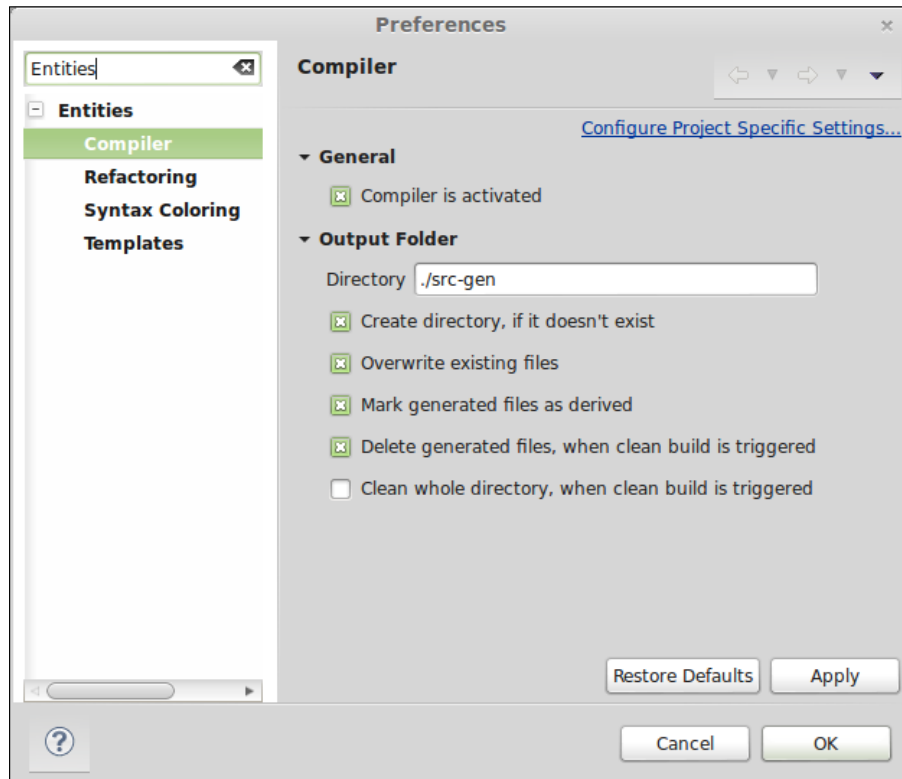
Standalone command-line compiler

We already know that the Xtext project wizard created the projects for our DSL separating the features related to the user interface in a separate project (the `.ui` project); the runtime project does not depend on the Eclipse user interface. Thus, we can create a command-line application consisting of a simple Java class with a `main` method. Simply add the following lines to your MWE2 workflow file:

```
// generator API
fragment = generator.GeneratorFragment {
  generateJavaMain = true
}
```

If you now run the workflow, you will find a `Main` class in the `src` folder of your project in the `org.example.entities.generator` package. As you may recall from *Chapter 2*, *Creating Your First Xtext Language,* files generated into the `src` folder are only generated once (if they do not exist) and thus you can safely add/modify the logic of the `Main` class. This is not required at this point; we will use the class as it was generated. You do not have to worry about not understanding everything that is done by the `Main` class at this point, this will be revealed in later chapters. For now it is enough to know that the generated `Main` class' `main` method accepts a command-line argument for the file to parse, validate, and generate code for.

Finally, you can also export a JAR file for your standalone Entities compiler; in Eclipse, an easy way to do that is given in the following steps:

1. Run the `Main.Java` file as a Java application (right-click and select **Run As | Java Application**). In the console view you can see that the application terminates with the error "Aborting: no path to EMF resource provided!" since you did not specify any command-line argument (but that is not the reason why we are creating this launch configuration).

2. From the **File** menu select **Export... | Java | Runnable JAR File**, then click on **Next**.

3. Select the launch configuration you created in step 1, specify the path of the exported JAR file (for example, `entities-compiler.jar`), and in the **Library handling** section, select **Package required libraries into generated JAR** (see the following screenshot) and click on **Finish**.
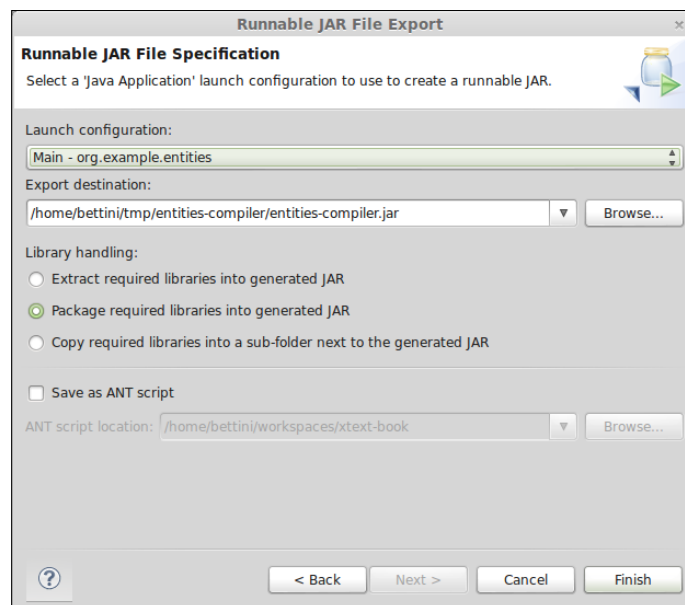
The generated JAR file is found in the directory denoted by the output path. Note that this JAR file is quite big (almost 20 MB) because besides the class files of your projects, it also contains all the required JARs (for example, the Xtext and EMF JAR files). This means that your JAR file is self-contained and does not need any further library. You can now try your standalone compiler from the command line, just go in the directory where the jar file has been created and run it with Java, giving the path of an `.entities` file as an argument:

```
java -jar entities-compiler.jar <path to an .entities file>
```

If the given file contains errors, you will see the errors reported as result; otherwise, you should see the message "Code generation finished". In the latter case, you will find all the generated Java files in the `src-gen` folder.
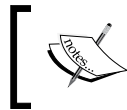
> This was a demonstration that Xtext can generate a standalone command-line based compiler that does not require a full Eclipse infrastructure.

Summary

Generating code from your DSL sources is a typical task when developing a DSL. Xtend provides many interesting features that make writing a code generator really easy.

Xtext automatically integrates your code generator into the Eclipse building infrastructure so that building takes place incrementally upon file saving, just like in the Eclipse JDT. You can also get a command-line standalone compiler so that your DSL programs can be compiled without Eclipse.

In the next chapter you will learn about the Dependency Injection framework Google Guice, on which Xtext heavily relies for customizing all of its features. In particular, you will also see how to customize the runtime and the IDE concepts for your DSL.

6

Customizations

In this chapter we describe the main mechanism for customizing Xtext components: Google Guice, a Dependency Injection framework. With Google Guice we can easily and consistently inject custom implementations of specific components into Xtext. In the first section, we will briefly show some Java examples that use Google Guice. Then, we will show how Xtext uses this dependency injection framework. In particular, you will learn how to customize both the runtime and the UI aspects.

Dependency injection

The Dependency Injection pattern (Fowler, 2004) is used to inject implementation classes into a class hierarchy in a consistent way. This is useful when classes delegate specific tasks to other classes: messages are simply forwarded to the objects referenced in fields (which abstract the actual behavior).

Let us consider a possible scenario: a `Service` class that abstracts from the actual implementation of a `Processor` class and a `Logger` class (although in this section we show Java code, all the injection principles naturally apply to Xtend as well). The following is a possible implementation:

```java
public class Service {
  private Logger logger;
  private Processor processor;

  public void execute(String command) {
    logger.log("executing " + command);
    processor.process(command);
    logger.log("executed " + command);
  }
}
```

```java
public class Logger {
  public void log(String message) {
    System.out.println("LOG: " + message);
  }
}

public interface Processor {
  public void process(Object o);
}

public class ProcessorImpl implements Processor {
  private Logger logger;

  public void process(Object o) {
    logger.log("processing");
    System.out.println("processing " + o + "...");
  }
}
```

These classes correctly abstract from the implementation details, but the problem of initializing the fields correctly still persists. If we initialize the fields in the constructor, then the user still needs to hardcode the actual implementation class names. Also note that Logger is used in two independent classes, thus, if we have a custom logger, we must make sure that all the instances use the correct one.

These issues can be dealt with by using dependency injection. With dependency injection hardcoded dependencies will be removed. Moreover, we will be able to easily and consistently switch the implementation classes throughout the code. Although the same goal can be achieved manually by implementing **factory method** or **abstract factory** patterns (Gamma et al., 1995), with dependency injection frameworks it is easier to keep the desired consistency and the programmer needs to write less code. Xtext uses the dependency injection framework Google Guice, `http://code.google.com/p/google-guice/`. We refer to the Google Guice documentation for all the features provided by this framework; in this section we just briefly describe its main features.

You annotate the fields you want Guice to inject with the `@Inject` annotation (`com.google.inject.Inject`):

```java
public class Service {
  @Inject private Logger logger;
  @Inject private Processor processor;

  public void execute(String command) {
    logger.log("executing " + command);
    processor.process(command);
```

```
      logger.log("executed " + command);
    }
  }

  public class ProcessorImpl implements Processor {
    @Inject private Logger logger;

    public void process(Object o) {
      logger.log("processing");
      System.out.println("processing " + o + "...");
    }
  }
```

The mapping from injection requests to instances is specified in a Guice Module,
a class that is derived from `AbstractModule`. The method `configure` is
implemented to specify the bindings using a simple and intuitive API.

You only need to specify the bindings for interfaces, abstract classes, and for custom
classes. This means that you do not need to specify a binding for `Logger` since it
is a concrete class; on the contrary, you need to specify a binding for the interface
`Processor`. The following is an example of a Guice module for our scenario:

```
  public class StandardModule extends AbstractModule {
    @Override
    protected void configure() {
      bind(Processor.class).to(ProcessorImpl.class);
    }
  }
```

You create an Injector using the static method `Guice.createInjector` by passing
a module. You then use the injector to create instances:

```
  Injector injector = Guice.createInjector(new StandardModule());
  Service service = injector.getInstance(Service.class);
  service.execute("First command");
```

The initialization of injected fields will be done automatically by Google Guice.
It is worth noting that the framework is also able to initialize (inject) private fields
(like in our example). Instances of classes that use dependency injection must
be created only through an injector; creating instances with `new` will not trigger
injection, thus all the fields annotated with `@Inject` will be null.

> When implementing a DSL with Xtext you will never have to
> create a new injector manually. In fact, Xtext generates utility
> classes to easily obtain an injector, for example, when testing your
> DSL with Junit, as we will see in *Chapter 7, Testing*. We also refer to
> Köhnlein, 2012 for more details. The example shown in this section
> only aims at presenting the main features of Google Guice.

If we need a different configuration of the bindings, all we need to do is define
another module. For example, let us assume that we have defined additional
derived implementations for logging and processing. Here is an example where
Logger and Processor are bound to custom implementations:

```
public class CustomModule extends AbstractModule {
  @Override
  protected void configure() {
    bind(Logger.class).to(CustomLogger.class);
    bind(Processor.class).to(AdvancedProcessor.class);
  }
}
```

Creating instances with an injector obtained using this module will ensure that the
right classes are used consistently (for example, the CustomLogger class will be used
both by Service and Processor).

You can create instances from different injectors in the same application, for example:

```
executeService(Guice.createInjector(new StandardModule()));
executeService(Guice.createInjector(new CustomModule()));

void executeService(Injector injector) {
  Service service = injector.getInstance(Service.class);
  service.execute("First command");
  service.execute("Second command");
}
```

> It is possible to request injection in many different ways, such as
> injection of parameters to constructors, using named instances,
> specification of default implementation of an interface, setter
> methods, and much more. In this book, we will mainly use
> injected fields.

Injected fields are instantiated only once when the class is instantiated. There are situations where this is not ideal; you may want to decide when you need an element to be instantiated, or you may need to create several instances from within method bodies. In such situations, instead of injecting an instance of the wanted type C, you inject a `com.google.inject.Provider<C>` instance, which has a `get` method that produces an instance of C.

For example:

```java
public class Logger {
  @Inject
  private Provider<Utility> utilityProvider;

  public void log(String message) {
    System.out.println("LOG: " + message + " - " +
      utilityProvider.get().m());
  }
}
```

Each time we create a new instance of `Utility` by using the injected `Provider` class. The nice thing is that to inject a custom implementation of `Utility`, you do not need to provide a custom `Provider`: you just bind the `Utility` class in the Guice module and everything will work as expected:

```java
public class CustomModule extends AbstractModule {
  @Override
  protected void configure() {
    bind(Logger.class).to(CustomLogger.class);
    bind(Processor.class).to(AdvancedProcessor.class);
    bind(Utility.class).to(CustomUtility.class);
  }
}
```
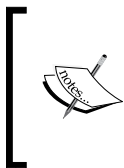
It is crucial to keep in mind that once classes rely on injection, their instances must be created only through an injector, otherwise all the injected elements will be null. In general, once dependency injection is used in a framework, all classes of the framework must rely on injection.

Google Guice in Xtext

All Xtext components rely on Google Guice dependency injection, even the classes that Xtext generates for your DSL. This means that in your classes, if you need to use a class from Xtext, you just have to declare a field of such type with the `@Inject` annotation.

The injection mechanism allows a DSL developer to customize basically every component of the Xtext framework.

When running the MWE2 workflow, Xtext generates both a fully configured module and an empty module derived from the generated one. Customizations are added to the empty stub module. The generated module should not be touched. Xtext generates one runtime module that defines the non-user interface related parts of the configuration and one specific for usage in the Eclipse IDE. Guice provides a mechanism for composing modules that is used by Xtext: the module in the UI project uses the module in the runtime project and overrides some bindings.

Let us consider the Entities DSL example; you can find in the `src` directory of the runtime project the class `EntitiesRuntimeModule`, which inherits from `AbstractEntitiesRuntimeModule` in the `src-gen` directory. Similarly, in the UI project you can find in the `src` directory the class `EntitiesUiModule`, which inherits from `AbstractEntitiesUiModule` in the `src-gen` directory.

The Guice modules in `src-gen` are already configured with the bindings for the stub classes generated during the MWE2 workflow. Thus, if you want to customize an aspect using a stub class then you do not have to specify any specific binding. The generated stub classes concern typical aspects that the programmer usually wants to customize, for example, validation and generation (as we saw in the previous chapters) in the runtime project, and labels and outline in the UI project (as we will see in the next sections). If you need to customize an aspect which is not covered by any of the generated stub classes, then you will need to write a class yourself and then specify the binding for your class in the Guice module in the `src` folder. We will see an example of this scenario in the section *Other customizations*.

Bindings in these Guice module classes can be specified as we saw in the previous section, by implementing the `configure` method. However, Xtext provides an enhanced API for defining bindings: Xtext reflectively searches for methods with a specific signature in order to find Guice bindings. Thus, assuming you want to bind a `BaseClass` class to your derived `CustomClass`, instead of writing the following:

```
public class EntitiesRuntimeModule extends ... {
  @Override
  protected void configure() {
    bind(BaseClass.class).to(CustomClass.class);
```

```
    ...

}
```

you can simply define a method in your module with a specific signature, as follows:

```
public Class<? extends BaseClass> bindBaseClass() {
  return CustomClass.class;
}
```

Remember that these methods are invoked reflectively, thus their signature must follow the expected convention. We refer to the official Xtext documentation for the complete description of the module API; typically, the binding methods that you will see in this book will have the preceding shape, in particular, the name of the method must start with "bind" followed by the name of the class or interface we want to provide a binding for.

It is important to understand that these bind methods do not necessarily have to override a method in the module base class; you can also make your own classes (which are not related to Xtext framework classes at all) participants of this injection mechanism (as long as you follow the preceding convention on method signatures). One important note is that bindings defined in a super module class must be overridden by a method with the same signature, or the result is an erroneous attempt to provide multiple/conflicting bindings of the same class/interface.

In the rest of this chapter we will show examples of customizations of both IDE and runtime concepts. For most of these customizations we will modify the corresponding Xtend stub class that Xtext generated when running the MWE2 workflow. As hinted before, in these cases we will not need to write a custom Guice binding. We will also show an example of a customization which does not have an automatically generated stub class.
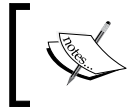
> Starting from version 2.4.0, all the stub classes generated by Xtext are Xtend classes. The only exceptions are the Guice module classes, which are Java classes.

Customizations of IDE concepts

In this section we show typical concepts of the IDE for your DSL that you may want to customize. Xtext shows its usability in this context as well, since, as you will see, it reduces the customization effort.

Labels

Xtext UI classes make use of an `ILabelProvider` interface to obtain textual labels and icons via its methods `getText` and `getImage`, respectively. `ILabelProvider` is a standard component of Eclipse JFace-based viewers. You can see an `ILabelProvider` interface in action in the Outline view and in content assist proposal pop-ups (as well as in various other places).

Xtext provides a default implementation of a label provider for all DSLs, which does its best to produce a sensible representation of the EMF model objects by using the `name` feature, if it is found in the corresponding object class, and a default image. You can see that in the Outline view when editing an `entities` file, refer to the following screenshot:
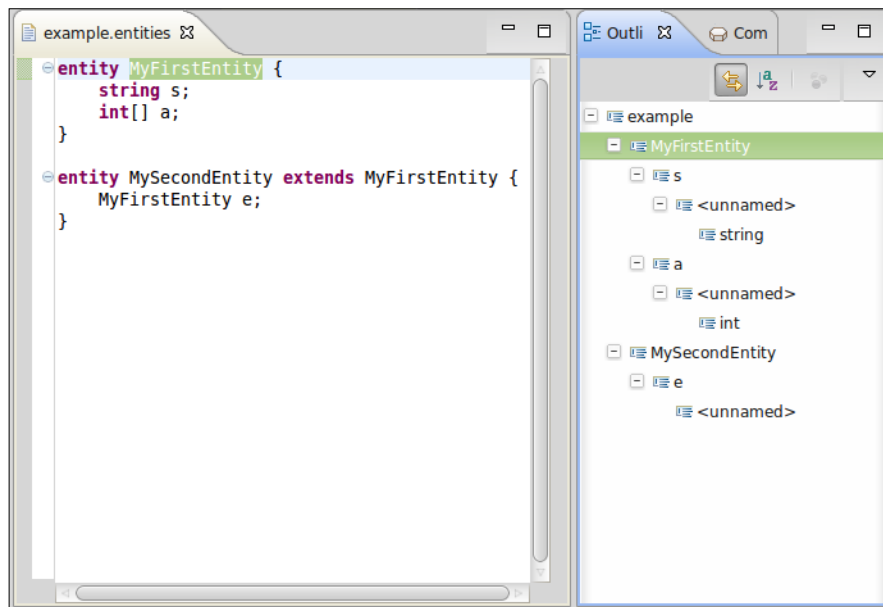
However, you surely want to customize the representation of some elements of your DSL.

The label provider stub class for your DSL can be found in the UI plug-in project in the subpackage `ui.labeling`. This stub class extends the base class `DefaultEObjectLabelProvider`. In the Entities DSL, the class is called `EntitiesLabelProvider`.

This class employs a **Polymorphic Dispatcher** mechanism (similar to the dispatch methods of Xtend described in *Chapter 3*, *The Xtend Programming Language*), which is also used in many other places in Xtext. Thus, instead of implementing the `getText` and `getImage` methods, you can simply define several versions of methods `text` and `image` taking as parameter an `EObject` object of the type you want to provide a representation for. Xtext will then search for such methods according to the runtime type of the elements to represent.

For example, for our Entities DSL we can change the textual representation of attributes in order to show their names and a better representation of types (for example, "name : type"). We then define a method `text` taking `Attribute` as a parameter and returning a string:

```
class EntitiesLabelProvider extends ... {

  @Inject extension TypeRepresentation

  def text(Attribute a) {
    a.name +
      if (a.type != null)
        " : " + a.type.representation
      else ""
  }
}
```

To get a representation of the `AttributeType` element, we use an injected extension, `TypeRepresentation` (it is similar to the way we generate Java code for attribute types in the generator of *Chapter 5*, *Code Generation*), in particular its method `representation`.

> Remember that the label provider is used, for example, for the Outline view, which is refreshed when the editor contents change, and its contents might contain errors; thus, you must be ready to deal with an incomplete model, and some features might still be null. That is why you should always check that the features are not null before accessing them.

Note that we inject an extension field of type `TypeRepresentation` instead of creating an instance with `new` in the field declaration. Although it is not strictly necessary to use injection for this class, we decided to rely on that because in the future, we might want to be able to provide a different implementation for that class. Another point for using injection instead of `new` is that the other class may rely on injection (or it may do so in the future). Using injection leaves the door open for future and unanticipated customizations.

The Outline now shows as in the following screenshot:
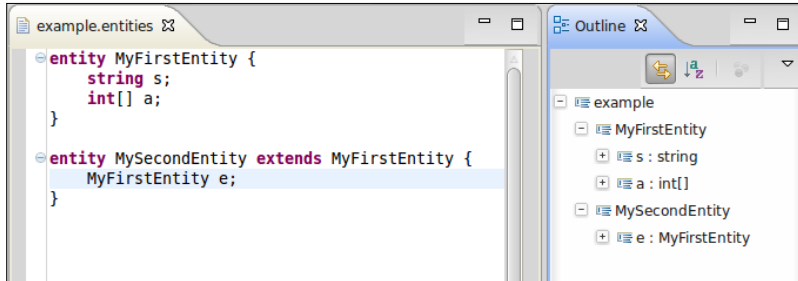
We can further enrich the labels for entities and attributes by using images for them. To do this, we create a directory where we place the image files of the icons we want to use. In order to benefit from Xtext's default handling of images, we call the directory `icons`, and we place two gif images there, `Entity.gif` and `Attribute. gif` (for entities and attributes respectively). We then define two `image` methods in `EntitesLabelProvider` where we only need to return the name of the image files (Xtext will do the rest for us):

```
class EntitiesLabelProvider extends DefaultEObjectLabelProvider {
  ... as before
  def image(Entity e) { "Entity.gif" }

  def image(Attribute a) { "Attribute.gif" }
}
```

You can see the result by relaunching Eclipse, as seen in the following screenshot:
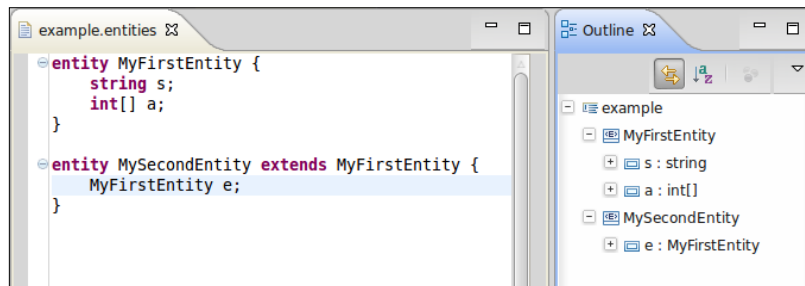
Now the entities and attributes labels look nicer.

If you plan to export the plugins for your DSL so that others can install them in their Eclipse (see *Chapter 11*, *Building and Releasing*), you must make sure that the icons directory is added to the build.properties file, otherwise that directory will not be exported. The bin.includes section of the build.properties file of your UI plugin should look like the following:

```
bin.includes = META-INF/,\
               .,\
               plugin.xml,\
               icons/
```

The Outline view

The default Outline view comes with nice features. In particular, it provides toolbar buttons to keep the Outline view selection synchronized with the element currently selected in the editor. Moreover, it provides a button to sort the elements of the tree alphabetically.

By default, the tree structure is built using the containment relations of the metamodel of the DSL; this strategy is not optimal in some cases. For example, an Attribute definition also contains the AttributeType element, which is a structured definition with children (for example, elementType, array, and length). This is reflected in the Outline (see the following screenshot) if you expand the Attribute elements.
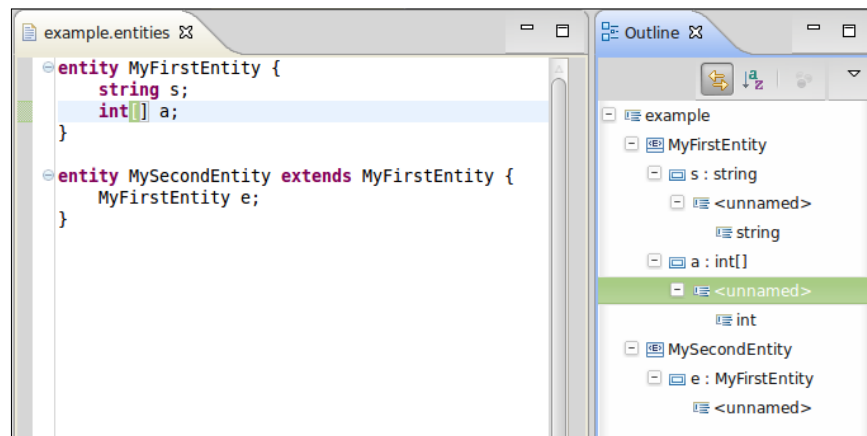
This shows unnecessary elements (such as `BasicType` names), which are now redundant since they are shown in the label of the attribute, and additional elements which are not representable with a name (such as the `array` feature).

We can influence the structure of the Outline tree by using the generated stub class `EntitiesOutlineTreeProvider` in the `src` folder `org.example.entities.ui.outline`. Also in this class, customizations are specified in a declarative way using the polymorphic dispatch mechanism. The official documentation details all the features that can be customized.

In our example we just want to make sure that the nodes for attributes are "leaf" nodes, that is, they cannot be further expanded and they have no children. In order to achieve this, we just need to define a method named `_isLeaf` (note the underscore) with a parameter of the type of the element, returning `true`. Thus, in our case we write:

```
class EntitiesOutlineTreeProvider extends
        DefaultOutlineTreeProvider {
   def _isLeaf(Attribute a) { true }
}
```

Let's relaunch Eclipse, and now see that the attribute nodes do not expose children anymore (see the following screenshot).
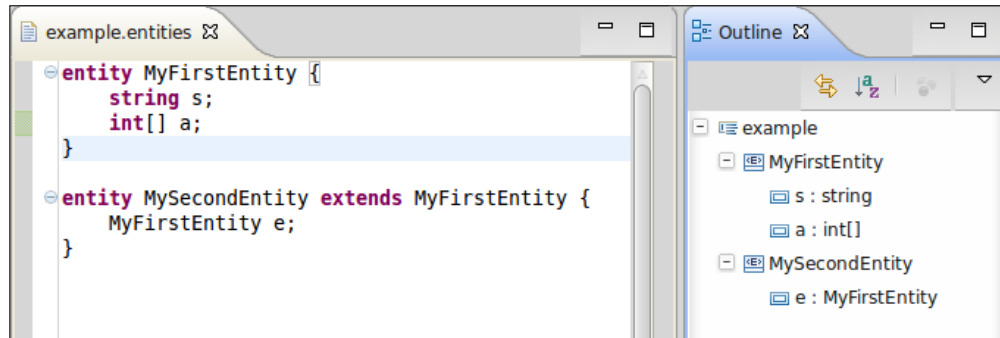
Besides defining leaf nodes, you can also specify the children in the tree for a specific node by defining a method `_createChildren` taking as parameters the type of the outline node and the type of the model element. This can be useful to define the actual root elements of the Outline tree. You see that, by default, the tree is rooted with a single node for the source file. In this example, it might be better to have a tree with many root nodes, each one representing an entity. The root of the Outline tree is always represented by a node of type `DefaultRootNode`; the root node is actually not visible, it is just the container of all nodes that will be displayed as roots in the tree.

Thus, we define the following method (our Entities model is rooted by a `Model` element):

```
public class EntitiesOutlineTreeProvider ... {
  ... as before
  def void _createChildren(DocumentRootNode outlineNode,
                            Model model) {
    model.entities.forEach[
      entity |
      createNode(outlineNode, entity);
    ]
  }
}
```

This way, when the Outline tree is built, we create a (root) node for each entity instead of having a single root for the source file (the method `createNode` is part of the Xtext base class); the result can be seen in the following screenshot:
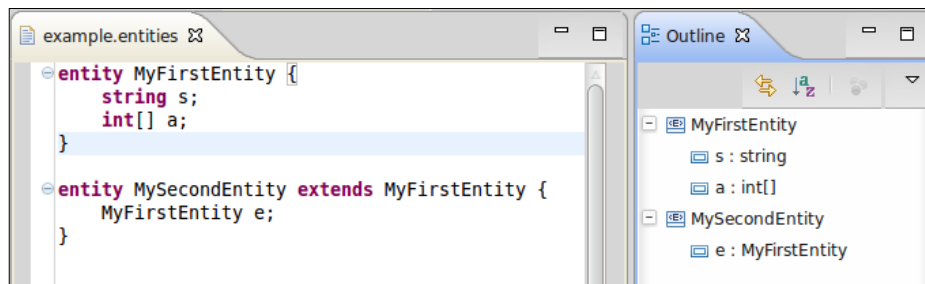
Customizing other aspects

In *Chapter 8*, *An Expression Language*, we will show how to customize the content assistant; there is no need to do this for the simple Entities DSL since the default implementation already does a fine job.

Custom formatting

If you tried to apply one of the quickfixes we implemented in *Chapter 4*, *Validation*, you might have noticed that after the EMF model has changed, the editor immediately reflects this change; however, the resulting textual representation is not well formatted (as shown in the following screenshot, where we applied the quickfix which adds the missing entity `FooBar`).
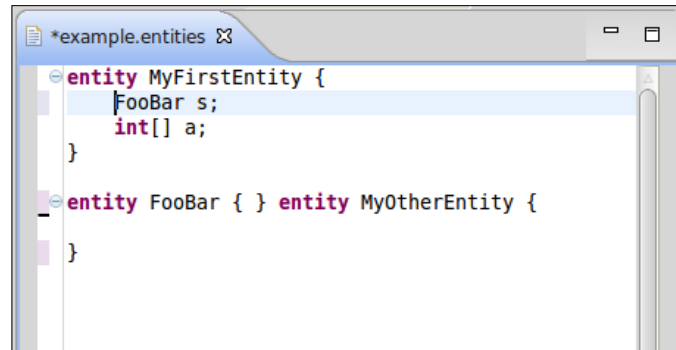
In general, the EMF model representing the AST does not contain any information about the textual representation, that is, all white space characters are not part of the EMF model (after all, the AST is an abstraction of the actual program).

Xtext keeps track of such information in another in-memory model called the **node model**. The node model carries the syntactical information, that is, offset and length in the textual document. However, when we manually change the EMF model, we do not provide any formatting directives, and Xtext uses the default formatter to get a textual representation of the modified or added model parts.

Indeed, in the editor of your DSL, Xtext already generated the menu for formatting your source; as it is standard in Eclipse editors (for example, the JDT editor), you can access the **Format** menu from the context menu of the editor or by using the key combination *Ctrl + Shift + F*.

The default formatter is `OneWhitespaceFormatter` and you can test this in the Entities DSL editor: this formatter simply separates all tokens of your program with a space. Typically you will want to change this default behavior.

If you provide a custom formatter, this will be used not only when the **Format** menu is invoked, but also when Xtext needs to update the editor contents after a manual modification of the AST model (for example, a quickfix performing a semantic modification).

The Xtext generator has created a stub class for customizing the formatting for your DSL. The formatter stub can be found in the main plugin project, and for our Entities DSL the class is `org.example.entities.formatting.EntitiesFormatter`.

The method to implement is `configureFormatting`, which is given a `FormattingConfig` object as an argument. Using this object we can declaratively state the formatting actions (for example, do a line wrap, indent, remove spaces, and so on) to be applied on specific tokens (for example, before, after, around, and so on).

For our Entities DSL, we decide to perform formatting as follows:

- Insert one line-wrap after the entity left curly bracket "{"
- Insert two line-wraps after the entity right curly bracket "}" so that entities will be separated by an empty line
- Indent tokens between entities curly brackets (so that attributes will be indented within an entity)
- Insert one line-wrap after each attribute declaration (that is, after the ";")
- Remove possible white spaces before the ";" of an attribute declaration

The `FormattingConfig` method names are descriptive (for example, `setIndentation`, `setLinewrap`, and so on). To specify the token to which we want to apply the formatting, we can rely on the `GrammarAccess` class of our DSL (generated by Xtext); in our example it is called `EntitiesGrammarAccess`. Using this class we can access the tokens related to rules (for example, `getEntityAccess`, `getAttributeAccess`, and so on) and keywords (for example, curly brackets, semicolons, and so on). For the latter, Eclipse content assist will help you in selecting the right method to use.

Summarizing, to achieve the formatting that we have just sketched, we need to implement the `configureFormatting` method as follows:

```
class EntitiesFormatter extends AbstractDeclarativeFormatter {

  @Inject EntitiesGrammarAccess g

  override protected void configureFormatting(FormattingConfig c){
    // formatting for entities
    val e = g.getEntityAccess()
    // indentation between { }
    c.setIndentation(e.getLeftCurlyBracketKeyword_3(),
                     e.getRightCurlyBracketKeyword_5())
    // newline after {
    c.setLinewrap().after(e.getLeftCurlyBracketKeyword_3())
```

```
        // two newlines after }
        c.setLinewrap(2).after(e.getRightCurlyBracketKeyword_5())
        // formatting for attributes
        val a = g.getAttributeAccess()
        // newline after ;
        c.setLinewrap(1).after(a.getSemicolonKeyword_2())
        // remove possible spaces before the ;
        c.setNoSpace().before(a.getSemicolonKeyword_2())
    }
}
```

If you now relaunch Eclipse and try to apply our quickfixes, you will see that the added entity is nicely formatted, as shown in the following screenshot:
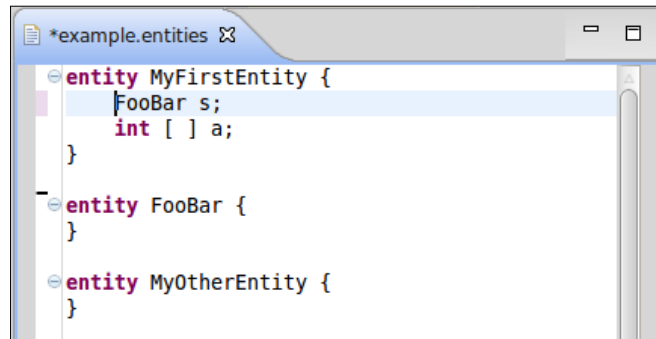
Moreover, you can also try to format the whole source in the editor by selecting **Format** from the context menu or by using the key combination *Ctrl + Shift + F*.

Other customizations

All the customizations you have seen so far were based on modification of a generated stub class with accompanying generated Guice bindings in the module under the `src-gen` directory.

However, since Xtext relies on injection everywhere, it is possible to inject a custom implementation for any mechanism, even if no stub class has been generated.

If you installed Xtext SDK in your Eclipse, the sources of Xtext are available for you to inspect. You should learn to inspect these sources by navigating to them and see what gets injected and how it is used. Then you are ready to provide a custom implementation and inject it. You can use the Eclipse **Navigate** menu. In particular, to quickly open a Java file (even from a library if it comes with sources), use *Ctrl* + *Shift* + *T* (**Open Type...**). This works only for Java classes, so if you want to quickly open another source file (for example, an Xtend class or the Xtext grammar file) use *Ctrl* + *Shift* + *R* (**Open Resource...**). Both dialogs have a text field where, if you start typing, the available elements soon show up. Eclipse supports **CamelCase** everywhere, so you can just type the capital letters of a compound name to quickly get to the desired element; for example, to open the `EntitiesRuntimeModule` Java class, use the **Open Type...** menu and just digit "ERM" to see the filtered results.

As an example, we show how to customize the output directory where the generated files will be stored (as we saw in *Chapter 5*, *Code Generation*, the default is `src-gen`). Of course, this output directory can be modified by the user using the Properties dialog that Xtext generated for your DSL (see *Chapter 5*, *Code Generation*), but we want to customize the default output directory for Entities DSL so that it becomes `entities-gen`.

The default output directory is retrieved internally by Xtext using an injected `IOutputConfigurationProvider` instance. If you take a look at this class (see the preceding tip), you will see:

```
import com.google.inject.ImplementedBy;

@ImplementedBy(OutputConfigurationProvider.class)
public interface IOutputConfigurationProvider {
  Set<OutputConfiguration> getOutputConfigurations();
  ...
```

The `@ImplementedBy` Guice annotation tells the injection mechanism the default implementation of the interface. Thus, what we need to do is create a subclass of the default implementation (that is, `OutputConfigurationProvider`) and provide a custom binding for the interface `IOutputConfigurationProvider`.

The method we need to override is `getOutputConfigurations`; if we take a look at its default implementation we see:

```
public Set<OutputConfiguration> getOutputConfigurations() {
  OutputConfiguration defaultOutput = new
    OutputConfiguration(IFileSystemAccess.DEFAULT_OUTPUT);
  defaultOutput.setDescription("Output Folder");
  defaultOutput.setOutputDirectory("./src-gen");
```

```
    defaultOutput.setOverrideExistingResources(true);
    defaultOutput.setCreateOutputDirectory(true);
    defaultOutput.setCleanUpDerivedResources(true);
    defaultOutput.setSetDerivedProperty(true);
    return newHashSet(defaultOutput);
}
```

Of course, the interesting part is the call to setOutputDirectory. We might simply copy the whole method body in our overridden version of the method and change that line; however, we prefer to:

1. Call the super implementation and store the result in a local variable (it is a set).
2. Take the first element of the set.
3. Call setOutputDirectory with our desired output directory.
4. Return the set stored in the local variable.

Instead of defining a Java subclass, we define an Xtend subclass. The preceding operations are easily implemented using the Xtend "with operator" (see *Chapter 3, The Xtend Programming Language*); moreover, we use implicit static extension methods from the Xtend default library (in particular, head, which returns the first element of the set) and the syntactic sugar for setter methods. Our custom Xtend class is as follows:

```
class EntitiesOutputConfigurationProvider extends
OutputConfigurationProvider {

    public val ENTITIES_GEN = "./entities-gen"

    override getOutputConfigurations() {
        super.getOutputConfigurations() => [
            head.outputDirectory = ENTITIES_GEN
        ]
    }
}
```

Note that we use a public constant for the output directory since we might need it later in other classes.

We create this class in the main plug-in project, since this concept is not just an UI concept (it is used also in other parts of the framework); moreover, since it deals with generation functionalities, we create it in the generator subpackage.

Now we must bind our implementation in the `EntitiesRuntimeModule` class:

```
public class EntitiesRuntimeModule extends
    AbstractEntitiesRuntimeModule {

  public Class<? extends IOutputConfigurationProvider>
        bindIOutputConfigurationProvider() {
    return EntitiesOutputConfigurationProvider.class;
  }
}
```

If we now relaunch Eclipse, we can verify that the Java code is generated into `entities-gen` instead of `src-gen`, as shown in the following screenshot. If you previously used the same project, the `src-gen` directory might still be there from previous generations; you need to manually remove it and set the new `entities-gen` as a source folder.
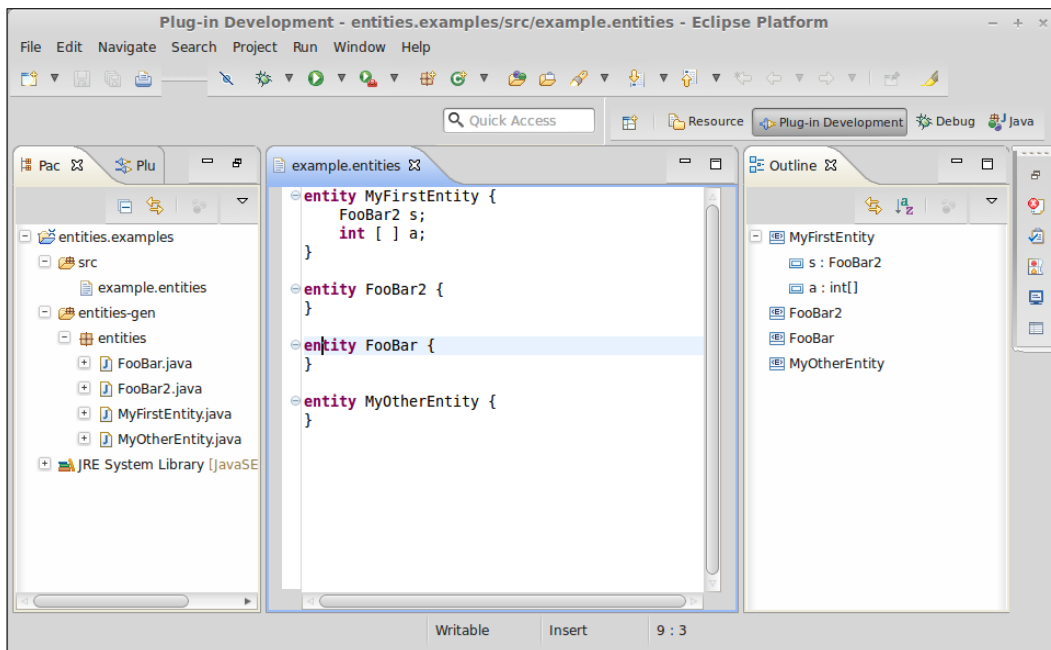
Summary

In this chapter we introduced the Google Guice dependency injection framework on which Xtext relies. You should now be aware of how easy it is to inject custom implementations consistently throughout the framework. You also learned how to customize some basic runtime and IDE concepts for a DSL.

The next chapter shows how to perform unit testing for languages implemented in Xtext. Test driven development is an important programming technique which will make your implementations more reliable, resilient to changes of the libraries, and will allow you to program quickly.

7
Testing

In this chapter you will learn how to test a DSL implementation by using the Junit framework and the additional utility classes provided by Xtext. This way, your DSL implementation will have a suite of tests that can be run automatically. We will use the Entities DSL developed in previous chapters for showing the typical techniques for testing both the runtime and the UI features of a DSL implemented in Xtext.

Introduction to testing

Writing automated tests is a fundamental technology/methodology when developing software. It will help you write quality software where most aspects (possibly all aspects) are somehow verified in an automatic and continuous way. Although successful tests do not guarantee that the software is bug free, automated tests are a necessary condition for professional programming (see Beck 2002, Martin 2002, 2008, 2011 for some insightful reading about this subject).

Tests will also document your code, whether it is a framework, a library, or an application; tests are a form of documentation that does not risk to get stale with respect to the implementation itself. Javadoc comments will likely not be kept in synchronization with the code they document, manuals will tend to become obsolete if not updated consistently, while tests will fail if they are not up-to-date.

The **Test Driven Development** (**TDD**) methodology fosters the writing of tests even before writing production code. When developing a DSL one can relax this methodology by not necessarily writing the tests first. However, one should write tests as soon as a new functionality is added to the DSL implementation. This must be taken into consideration right from the beginning, thus, you should not try to write the complete grammar of a DSL, but proceed gradually; write a few rules to parse a minimal program, and immediately write tests for parsing some test input programs. Only when these tests pass you should go on to implementing other parts of the grammar.

Moreover, if some validation rules can already be implemented with the current version of the DSL, you should write tests for the current validator checks as well.

Ideally, one does not have to run Eclipse to manually check whether the current implementation of the DSL works as expected. Using tests will then make the development much faster.

The number of tests will grow as the implementation grows, and tests should be executed each time you add a new feature or modify an existing one. You will see that since tests will run automatically, executing them over and over again will require no additional effort besides triggering their execution (think instead if you should manually check that what you added or modified did not break something).

This also means that you will not be scared to touch something in your implementation; after you did some changes, just run the whole test suite and check whether you broke something. If some tests fail, you will just need to check whether the failure is actually expected (and in case fix the test) or whether your modifications have to be fixed.
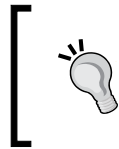
> It is worth noting that using a version control system (such as Git) is essential to easily get back to a known state; just experimenting with your code and finding errors using tests does not mean you can easily backtrack.

You will not even be scared to port your implementation to a new version of the used frameworks. For example, when a new version of Xtext is released, it is likely that some API has changed and your DSL implementation might not be built anymore with the new version. Surely, running the MWE2 workflow is required. But after your sources compile again, your test suite will tell you whether the behavior of your DSL is still the same. In particular, if some of the tests fail, you can get an immediate idea of which parts need to be changed to conform to the new version of Xtext.

Moreover, if your implementation relies on a solid test suite, it will be easier for contributors to provide patches and enhancements for your DSL; they can run the test suite themselves or they can add further tests for a specific bugfix or for a new feature. It will also be easy for the main developers to decide whether to accept the contributions by running the tests.

Last but not the least, you will discover that writing tests right from the beginning will force you to write modular code (otherwise you will not be able to easily test it) and it will make programming much more fun.

Xtext and Xtend themselves are developed with a test driven approach.

Junit 4

Junit is the most popular unit test framework for Java and it is shipped with the Eclipse JDT. In particular, the examples in this book are based on Junit version 4.

To implement Junit tests, you just need to write a class with methods annotated with `@org.junit.Test`. We will call such methods simply **test methods**. Such Java (or Xtend) classes can then be executed in Eclipse using the "Junit test" launch configuration; all methods annotated with `@Test` will be then executed by Junit. In test methods you can use **assert** methods provided by Junit to implement a test. For example, `assertEquals(expected, actual)` checks whether the two arguments are equal; `assertTrue(expression)` checks whether the passed expression evaluates to true. If an assertion fails, Junit will record such failure; in particular, in Eclipse, the Junit view will provide you with a report about tests that failed. Ideally, no test should fail (and you should see the green bar in the Junit view).

> All test methods can be executed by Junit in any order, thus, you should never write a test method which depends on another one; all test methods should be executable independently from each other.

If you annotate a method with `@Before`, that method will be executed before each test method in that class, thus, it can be used to prepare a common setup for all the test methods in that class. Similarly, a method annotated with `@After` will be executed after each test method (even if it fails), thus, it can be used to cleanup the environment. A static method annotated with `@BeforeClass` will be executed only once before the start of all test methods (`@AfterClass` has the complementary intuitive functionality).

The ISetup interface

Running tests means we somehow need to bootstrap the environment to make it support EMF and Xtext in addition to the implementation of our DSL. This is done with a suitable implementation of `ISetup`. We need to configure things differently depending on how we want to run tests; with or without Eclipse and with or without Eclipse UI being present. The way to set up the environment is quite different when Eclipse is present, since many services are shared and already part of the Eclipse environment. When setting up the environment for non-Eclipse use (also referred to as **standalone**) there are a few things that must be configured, such as creating a Guice injector and registering information required by EMF. The method `createInjectorAndDoEMFRegistration` in the `ISetup` interface is there to do exactly this.

Besides the creation of an `Injector`, this method also performs all the initialization of EMF global registries so that after the invocation of that method, the EMF API to load and store models of your language can be fully used, even without a running Eclipse. Xtext generates an implementation of this interface, named after your DSL, which can be found in the runtime plugin project. For our Entities DSL it is called `EntitiesStandaloneSetup`.
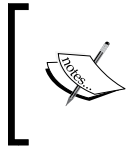
> The name "standalone" expresses the fact that this class has to be used when running outside Eclipse. Thus, the preceding method must never be called when running inside Eclipse (otherwise the EMF registries will become inconsistent).

In a plain Java application the typical steps to set up the DSL (for example, our Entities DSL) can be sketched as follows:

```
Injector injector = new EntitiesStandaloneSetup().
createInjectorAndDoEMFRegistration();
XtextResourceSet resourceSet =
  injector.getInstance(XtextResourceSet.class);
resourceSet.addLoadOption
  (XtextResource.OPTION_RESOLVE_ALL, Boolean.TRUE);
Resource resource = resourceSet.getResource
  (URI.createURI("/path/to/my.entities"), true);
Model model = (Model) resource.getContents().get(0);
```

This standalone setup class is especially useful also for Junit tests that can then be run without an Eclipse instance. This will speed up the execution of tests. Of course, in such tests you will not be able to test UI features.

As we will see in this chapter, Xtext provides many utility classes for testing which do not require us to set up the runtime environment explicitly. However, it is important to know about the existence of the setup class in case you either need to tweak the generated standalone compiler or you need to set up the environment in a specific way for unit tests.

Implementing tests for your DSL

Xtext highly fosters using unit tests, and this is reflected by the fact that, by default, the MWE2 workflow generates a specific plug-in project for testing your DSL. In fact, usually tests should reside in a separate project, since they should not be deployed as part of your DSL implementation. This additional project ends with the `.tests` suffix, thus, for our Entities DSL, it is `org.example.entities.tests`. The tests plug-in project has the needed dependencies on the required Xtext utility bundles for testing.

We will use Xtend to write Junit tests.

In the `src-gen` directory of the tests project, you will find the injector providers for both headless and UI tests. You can use these providers to easily write Junit test classes without having to worry about the injection mechanisms setup. The Junit tests that use the injector provider will typically have the following shape (using the Entities DSL as an example):

```
@RunWith(typeof(XtextRunner))
@InjectWith(typeof(EntitiesInjectorProvider))
class MyTest {
  @Inject MyClass

  ...
```

As hinted in the preceding code, in this class you can rely on injection; we used `@InjectWith` and declared that `EntitiesInjectorProvider` has to be used to create the injector. `EntitiesInjectorProvider` will transparently provide the correct configuration for a standalone environment. As we will see later in this chapter, when we want to test UI features, we will use `EntitiesUiInjectorProvider` (note the "`Ui`" in the name).

Testing the parser

The first tests you might want to write are the ones which concern parsing.

This reflects the fact that the grammar is the first thing you must write when implementing a DSL. You should not try to write the complete grammar before starting testing: you should write only a few rules and soon write tests to check if those rules actually parse an input test program as you expect.

The nice thing is that you do not have to store the test input in a file (though you could do that); the input to pass to the parser can be a string, and since we use Xtend, we can use multi-line strings.

The Xtext test framework provides the class `ParseHelper` to easily parse a string. The injection mechanism will automatically tell this class to parse the input string with the parser of your DSL. To parse a string, we inject an instance of `ParseHelper<T>`, where `T` is the type of the root class in our DSL's model – in our Entities example, this class is called `Model`. The method `ParseHelper.parse` will return an instance of `T` after parsing the input string given to it.

By injecting the `ParseHelper` class as an extension, we can directly use its methods on the strings we want to parse. Thus, we can write:

```
@RunWith(typeof(XtextRunner))
@InjectWith(typeof(EntitiesInjectorProvider))
class EntitiesParserTest {

  @Inject extension ParseHelper<Model>

  @Test
  def void testParsing() {
    val model = '''
      entity MyEntity {
          MyEntity attribute;
       }
     '''.parse

    val entity = model.entities.get(0)
    Assert::assertEquals("MyEntity", entity.name)

    val attribute = entity.attributes.get(0)
    Assert::assertEquals("attribute", attribute.name);
    Assert::assertEquals("MyEntity",
      (attribute.type.elementType as EntityType).
        entity.name);
  }
  ...
```

In this test, we parse the input and test that the expected structure was constructed as a result of parsing. These tests do not add much value in the Entities DSL, but in a more complex DSL you do want to test that the structure of the parsed EMF model is as you expect (we will see an example of that in *Chapter 8*, *An Expression Language*).

You can now run the test: right-click on the Xtend file and select **Run As** | **JUnit Test** as shown in the following screenshot. The test should pass and you should see the green bar in the Junit view.
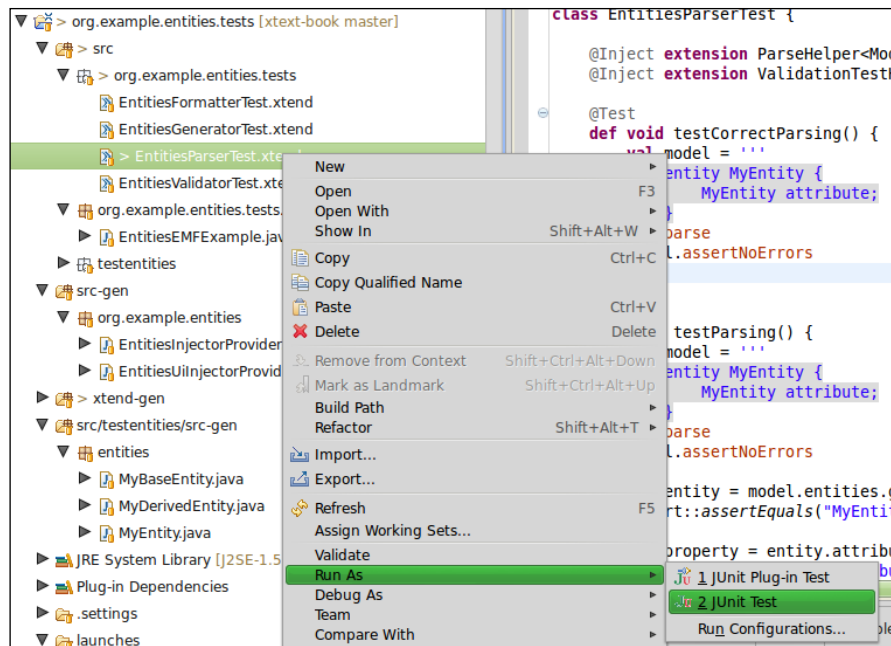
Note that the parse method returns an EMF model even if the input string contains syntax errors (it tries to parse as much as it can); thus, if you want to make sure that the input string is parsed without any syntax error, you have to check that explicitly. To do that, you can use another utility class, `ValidationTestHelper`. This class provides many assert methods that take an `EObject` argument. You can use an extension field and simply call `assertNoErrors` on the parsed EMF object. Alternatively, if you do not need the EMF object but you just need to check that there are no parsing errors, you can simply call it on the result of parse, for example:

```
class EntitiesParserTest {

  @Inject extension ParseHelper<Model>
  @Inject extension ValidationTestHelper
...
  @Test
  def void testCorrectParsing() {
    '''
      entity MyEntity {
          MyEntity attribute
      }
    '''.parse.assertNoErrors
  }
```

If you try to run the tests again, you will get a failure for this new test, as shown in the following screenshot:
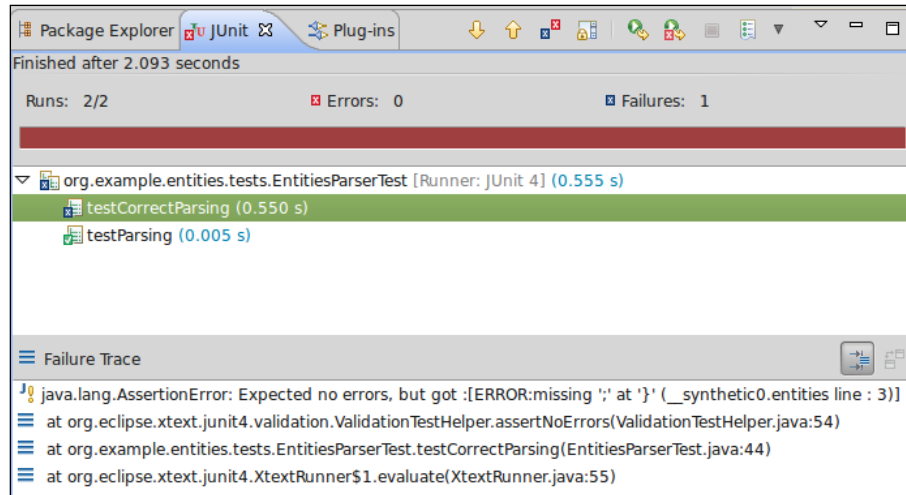
The reported error should be clear enough: we forgot to add the terminating ";" in our input program, thus we can fix it and run the test again; this time the green bar should be back.

You can now write other `@Test` methods for testing the various features of the DSL (see the sources of the examples). Depending on the complexity of your DSL you may have to write many of them.

> Tests should test one specific thing at a time; lumping things together (to reduce the overhead of having to write many test methods) usually makes it harder later.

Remember that you should follow this methodology while implementing your DSL, not after having implemented all of it. If you follow this strictly, you will not have to launch Eclipse to manually check that you implemented a feature correctly, and you will note that this methodology will let you program really fast.

Ideally, you should start with the grammar with a single rule, especially if the grammar contains nonstandard terminals. The very first task is to write a grammar that just parses all terminals. Write a test for that to ensure there are no overlapping terminals before proceeding; this is not needed if terminals are not added to the standard terminals. After that add as few rules as possible in each round of development/testing until the grammar is complete.

Testing the validator

Earlier we used the `ValidationTestHelper` class to test that it was possible to parse without errors. Of course, we also need to test that errors and warnings are detected. In particular, we should test any error situation handled by our own validator. The `ValidationTestHelper` class contains utility methods (besides `assertNoErrors`) that allow us to test whether the expected errors are correctly issued.

For instance, for our Entities DSL, we wrote a custom validator method that checks that the entity hierarchy is acyclic (*Chapter 4*, *Validation*). Thus, we should write a test that, given an input program with a cycle in the hierarchy, checks that such an error is indeed raised during validation.

Although not strictly required, it is better to separate Junit test classes according to the tested features, thus, we write another Junit class, `EntitiesValidatorTest`, which contains tests related to validation. The start of this new Junit test class should look familiar:

```
@RunWith(typeof(XtextRunner))
@InjectWith(typeof(EntitiesInjectorProvider))
class EntitiesValidatorTest {

  @Inject extension ParseHelper<Model>
  @Inject extension ValidationTestHelper
  ...
```

We are now going to use the `assertError` method from `ValidationTestHelper`, which, besides the EMF model element to validate, requires the following arguments:

- `EClass` of the object which contains the error (which is usually retrieved through the EMF `EPackage` class generated when running the MWE2 workflow)
- The expected Issue Code
- An optional string describing the expected error message

Thus, we parse input containing an entity extending itself and we pass
the arguments to `assertError` according to the error generated by
`checkNoCycleInEntityHierarchy` in `EntitiesValidator`
(see *Chapter 4, Validation*):

```
@Test
def void testEntityExtendsItself() {
  '''
    entity MyEntity extends MyEntity {

    }
  '''.parse.assertError(EntitiesPackage::eINSTANCE.entity,
    EntitiesValidator::HIERARCHY_CYCLE,
    "cycle in hierarchy of entity 'MyEntity'"
  )
}
```

Note that the `EObject` argument is the one returned by the `parse` method (we use
`assertError` as an extension method). Since the error concerns an `Entity` object, we
specify the corresponding `EClass` (retrieved using `EntitiesPackage`), the expected
Issue Code, and finally, the expected error message. This test should pass.

We can now write another test which tests the same validation error on a more
complex input with a cycle in the hierarchy involving more than one entity; in this
test we make sure that our validator issues an error for each of the entities involved
in the hierarchy cycle:

```
@Test
def void testCycleInEntityHierarchy() {
  val model = '''
    entity A extends B {}
    entity B extends C {}
    entity C extends A {}
  '''.parse

  model.assertError(EntitiesPackage::eINSTANCE.entity,
    EntitiesValidator::HIERARCHY_CYCLE,
    "cycle in hierarchy of entity 'A'"
  )
  model.assertError(EntitiesPackage::eINSTANCE.entity,
    EntitiesValidator::HIERARCHY_CYCLE,
    "cycle in hierarchy of entity 'B'"
  )
```

```
    model.assertError(EntitiesPackage::eINSTANCE.entity,
      EntitiesValidator::HIERARCHY_CYCLE,
      "cycle in hierarchy of entity 'C'"
    )
  }
```

Note that this time we must store the parsed EMF model into a variable since we will call `assertError` many times.

We can also test that the `NamesAreUniqueValidator` method detects elements with the same name (we refer to *Chapter 4*, *Validation*, for the details about this validator):

```
  @Test
  def void testDuplicateEntities() {
    val model = '''
      entity MyEntity {}

      entity MyEntity {}
    '''.parse

    model.assertError(EntitiesPackage::eINSTANCE.entity,
      null,
      "Duplicate Entity 'MyEntity'"
    )
  }
```

In this case, we pass `null` for the issue argument, since no Issue Code is reported by `NamesAreUniqueValidator`.

Similarly, we can write a test where the input has two attributes with the same name:

```
  @Test
  def void testDuplicateAttributes() {
    val model = '''
      entity MyEntity {
          MyEntity attribute;
          MyEntity attribute;
      }
    '''.parse

    model.assertError(EntitiesPackage::eINSTANCE.attribute,
      null,
      "Duplicate Attribute 'attribute'"
    )
  }
```

Note that in this test we pass the `EClass` corresponding to `Attribute`, since duplicate attributes are involved in the expected error.

> Do not worry if it seems tricky to get the arguments for `assertError` right the first time; writing a test that fails the first time it is executed is expected in Test Driven Development. The error of the failing test should put you on the right track to specify the arguments correctly. However, by inspecting the error of the failing test, you must first make sure that the actual output is what you expected, otherwise something is wrong either with your test or with the implementation of the component that you are testing.

Testing the formatter

As we said in the previous chapter, the formatter is also used in a non-UI environment (indeed, we implemented that in the runtime plug-in project), thus, we can test the formatter for our DSL with plain Junit tests. At the moment, there is no helper class in the Xtext framework for testing the formatter, thus we need to do some additional work to set up the tests for the formatter. This example will also provide some more details on Xtext and EMF, and it will introduce unit test methodologies that are useful in many testing scenarios where you need to test whether a string output is as you expect.

First of all, we create another Junit test class for testing the formatter; this time we do not need the helper for the validator; we will inject `INodeModelFormatter` as an extension field since this is the class internally used by Xtext to perform formatting.

> One of the main principles of unit testing (which is also its main strength) is that you should test a single functionality in isolation. Thus, to test the formatter, we must not run a UI test that opens an Xtext editor on an input file and call the menu item which performs the formatting; we just need to test the class to which the formatting is delegated and we do not need a running Eclipse for that.

```
import static extension org.junit.Assert.*

@RunWith(typeof(XtextRunner))
@InjectWith(typeof(EntitiesInjectorProvider))
class EntitiesFormatterTest {

  @Inject extension ParseHelper<Model>
  @Inject extension INodeModelFormatter;
```

Note that we import all the static methods of the Junit `Assert` class as extension methods.

Then, we write the code that actually performs the formatting given an input string. Since we will write several tests for formatting, we isolate such code in a reusable method. This method is not annotated with `@Test`, thus it will not be automatically executed by Junit as a test method. This is the Xtend code that returns the formatted version of the input string:

```
(input.parse.eResource as XtextResource).parseResult.
            rootNode.format(0, input.length).formattedText
```

The method `ParseHelper.parse` returns the EMF model object, and each `EObject` has a reference to the containing EMF resource; we know that this is actually `XtextResource` (a specialized version of an EMF resource). We retrieve the result of parsing, that is, an `IParseResult` object, from the resource. The result of parsing contains the `node` model; recall from *Chapter 6*, *Customizations*, that the node model carries the syntactical information that is, offsets and spaces of the textual input. The root of the node model, `ICompositeNode`, can be passed to the formatter to get the formatted version (we can even specify to format only a part of the input program).

Now we can write a reusable method that takes an input char sequence and an expected char sequence and tests that the formatted version of the input program is equal to what we expect:

```
def void assertFormattedAs(CharSequence input,
                           CharSequence expected) {
    expected.toString.assertEquals(
        (input.parse.eResource as XtextResource).parseResult.
            rootNode.format(0, input.length).formattedText)
}
```

The reason why we convert the expected char sequence into a string will be clear in a minute. Note the use of `Assert.assertEquals` as an extension method.

We can now write our first formatting test using our extension method `assertFormattedAs`:

```
@Test
def void testEntities() {
  '''
  entity E1 { } entity E2 {}
  '''.assertFormattedAs(
  '''...'''
  )
}
```

Why did we specify "…" as the expected formatted output? Why did we not try to specify what we really expect as the formatted output? Well, we could have written the expected output, and probably we would have gotten it right on the first try, but why not simply make the test fail and see the actual output? We can then copy that in our test once we are convinced that it is correct. So let's run the test, and when it fails, the Junit view tells us what the actual result is, as shown in the following screenshot:
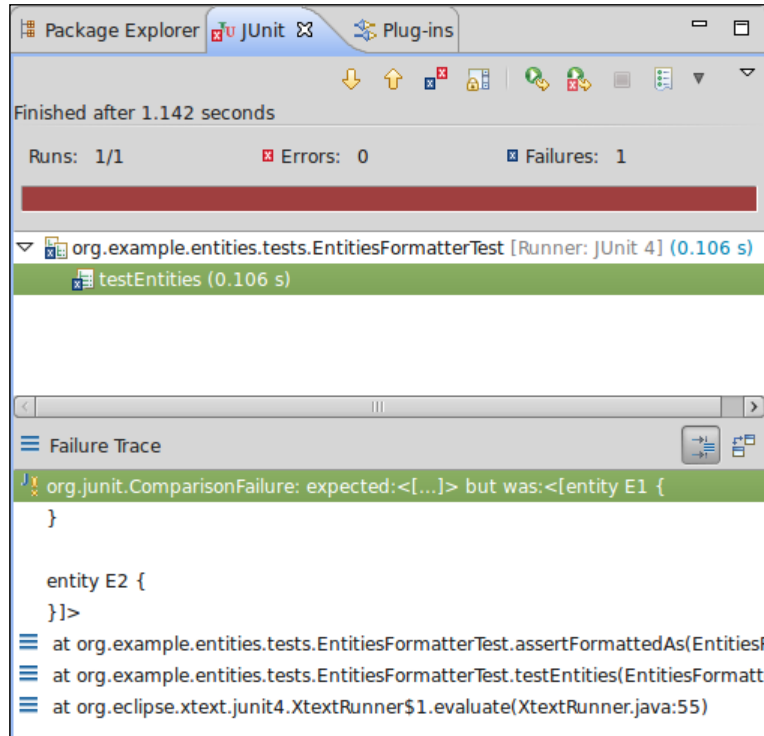
If you now double-click on the line showing the comparison failure in the Junit view, you will get a dialog showing a line by line comparison, as shown in the following screenshot:
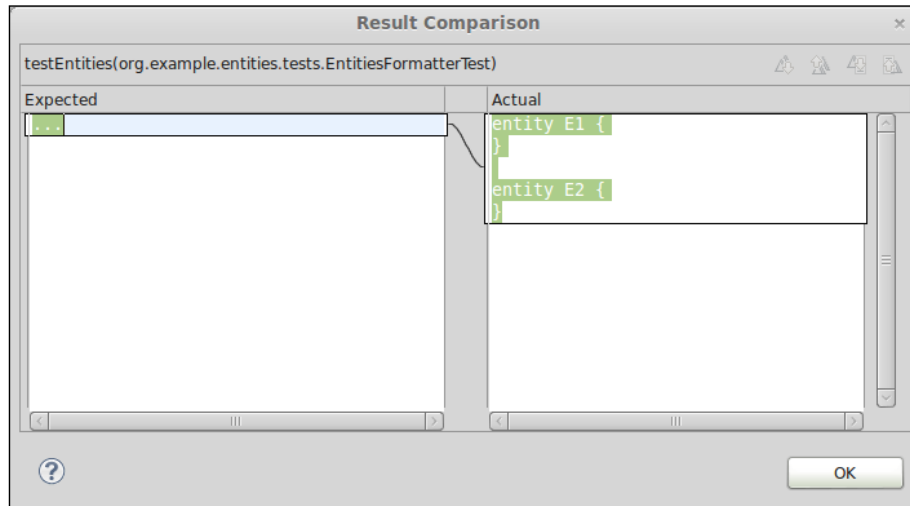
You can verify that the actual output is correct, copy that, and paste it into your test as the expected output. The test will now succeed:

```
@Test
def void testEntities() {
  '''
  entity E1 { } entity E2 {}
  '''.assertFormattedAs(
'''
entity E1 {
}

entity E2 {
}'''
    )
}
```

> We did not indent the expected output in the multi-line string since it is easy to paste it like that from the Junit dialog.

Using this technique you can easily write Junit tests that deal with comparisons. However, the "Result Comparison" dialog appears only if you pass `String` objects to `assertEquals`; that is why we converted the char sequence into a string in the implementation of `assertFormattedAs`.

We now add a test for testing the formatting of attributes; the final result will be:

```
@Test
def void testAttributes() {
    '''
    entity E1 { int i ; string s; boolean b    ;}
    '''.assertFormattedAs(
'''
    entity E1 {
    int i;
    string s;
    boolean b;
}'''
    )
}
```

Testing code generation

Xtext provides a helper class to test your code generator, `CompilationTestHelper`, which we inject as an extension field in the Junit test class. This helper class parses an input string, validates the model, and runs the code generator, thus we do not need the parser helper in this test class:

```
@RunWith(typeof(XtextRunner))
@InjectWith(typeof(EntitiesInjectorProvider))
class EntitiesGeneratorTest {

  @Inject extension CompilationTestHelper
```

This helper class provides the method `assertCompilesTo`, which takes a char sequence representing an input program and a char sequence representing the expected generated output. Using that as an extension method, we can then write the following test method, which tests that the generated Java code is as we expect (we use the technique of the Junit view to get the actual output as illustrated in the previous section):

```
@Test
def void testGeneratedCode() {
  '''
  entity MyEntity {
    string myAttribute;
  }
  '''.assertCompilesTo(
'''
package entities;

public class MyEntity {
  private String myAttribute;

  public String getMyAttribute() {
    return myAttribute;
  }

  public void setMyAttribute(String _arg) {
  this.myAttribute = _arg;
  }

}
''')
}
```

Testing that the generated output corresponds to what we expect is already a good testing technique. However, when the generated code is Java code (as in our Entities DSL), it might be good to also test that it is valid Java code (that is, the Java compiler compiles the generated code without errors). In a simple DSL like Entities, having a look at the generated Java code might be enough to be convinced that it is valid Java code, but, in a more involved DSL, manually checking that the (probably complex) generated Java code is valid is not an option.

To test that the generated code is valid Java code, we use the `CompilationTestHelper.compile` method, which takes an input string and a lambda. The parameter of the lambda is a `Result` object (an inner class). In the lambda we can use the `Result` object to perform additional checks. To test the validity of the generated Java code, we can call the `Result.getCompiledClass` method. This method compiles the generated code with the Eclipse Java compiler. If the Java compiler issues an error, then our test will fail and the Junit view will show the compilation errors.

> Since the Eclipse JDT compiler is used to test the compilation, you will need to add `org.eclipse.jdt.core` as a required bundle in your `.tests` plug-in project's `MANIFEST.MF` file.

We write the following test (remember that if no parameter is explicitly declared in the closure, the special implicit variable `it` is used):

```
@Test
def void testGeneratedValidJavaCode() {
  '''
  entity MyEntity {
    string myAttribute;
  }
  '''.compile[getCompiledClass]
  // check that it is valid Java code
}
```

If the `getCompiledClass` class terminates successfully, it also returns a Java `Class` object which can be used to instantiate the compiled Java class by reflection; this allows us to test the generated Java class' runtime behavior. We can easily invoke the created instance's methods via the reflection support provided by the Xtext class `ReflectExtensions`. For example:

```
…
@Inject extension ReflectExtensions

@Test
def void testGeneratedJavaCode() {
  '''
  entity E {
    string myAttribute;
  }
```

```
  '''.compile[
    getCompiledClass.newInstance => [
      assertNull(it.invoke("getMyAttribute"))
      it.invoke("setMyAttribute", "value")
      assertEquals("value",
        it.invoke("getMyAttribute"))
    ]
  ]
}
```

This method tests (via the getter method) that the attributes are initialized to `null` and that the setter method sets the corresponding attribute.

You can test situations when the generator generates several files originating from a single input. In the Entities DSL, a Java class is generated for each entity, thus, we can perform checks on each generated Java file by using the method `Result.getGeneratedCode(String)` that takes the name of the generated Java class as an argument and returns its contents. Since the generator for our Entities DSL should generate the Java class `entities.E` for an entity named `"E"`, we can test this with the following test:

```
@Test def void testGeneratedCodeWithTwoEntites() {
  '''
  entity FirstEntity {
    SecondEntity myAttribute;
  }

  entity SecondEntity { }
  '''.compile[
'''
package entities;

public class FirstEntity {
  private SecondEntity myAttribute;

  public SecondEntity getMyAttribute() {
    return myAttribute;
  }

  public void setMyAttribute(SecondEntity _arg) {
    this.myAttribute = _arg;
  }

}
```

```
'''.
toString.assertEquals(getGeneratedCode("entities.FirstEntity"))

'''
package entities;

public class SecondEntity {

}
'''.
toString.assertEquals(getGeneratedCode("entities.SecondEntity"))
  ]
}
```
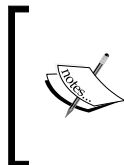
> The method `getGeneratedCode` assumes that the requested generated artifact is a Java file. If your DSL also generates other artifacts, for example, XML files, then you will have to write a custom `CompilationTestHelper` class if you want to test the generated artifacts which are not Java files.

Similarly, we can check that the several generated Java files compile correctly and we can perform reflective Java operations on all the compiled Java classes using `Result.getGeneratedCode(String)` specifying the fully qualified name of the generated Java class:

```
@Test def void testGeneratedJavaCodeWithTwoClasses() {
  '''
  entity FirstEntity {
    SecondEntity myAttribute;
  }

  entity SecondEntity {
    string s;
  }
'''.compile[
    val FirstEntity =
      getCompiledClass("entities.FirstEntity").newInstance
    val SecondEntity =
      getCompiledClass("entities.SecondEntity").newInstance
    SecondEntity.invoke("setS", "testvalue")
    FirstEntity.invoke("setMyAttribute", SecondEntity)
```

```
    SecondEntity.assertSame(FirstEntity.invoke("getMyAttribute"))
    "testvalue".assertEquals
      (FirstEntity.invoke("getMyAttribute").invoke("getS"))
    ]
}
```

In particular, in this last example, the first generated Java class depends on the second generated Java class.

Again, these tests might not be valuable in this DSL, but in more complex DSLs, having tests which automatically check the runtime behavior of the generated code enhances productivity.

Test suite

When you write several Junit classes, it becomes uncomfortable to run them individually; after all, you write unit tests because you want an automatic mechanism to test your implementation, and you want to run all tests with just one operation.

When Xtext first generates the projects for your DSL in the `.tests` plug-in project, it also generates a Junit launch which automatically runs all the Junit tests in that project. This launch can be found in the root folder of the test plug-in project; in our Entities DSL it is called `org.example.entities.tests.launch`. You can execute this launch by going to **Run As | Junit Test**.

If you need more control over the tests that must be run or you want to group some tests, you can write a Junit **Test Suite**. For example, you can write a suite for tests which are not related to generation as follows:

```java
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses({
    EntitiesParserTest.class,
    EntitiesFormatterTest.class,
    EntitiesValidatorTest.class
})
public class EntitiesNotGeneratorRelatedTests {
}
```

You can run such suite as a standard Junit test and it will run all the test methods in all the test classes specified.

Testing the UI

Most of the mechanisms of a DSL implemented in Xtext can be tested with plain Java Junit tests without a UI environment. However, when testing UI features, tests need a running Eclipse.

In the Entities DSL we did not customize the content assist, thus we do not really need to test it; however, for more complex DSLs, you want to test that the custom content assist works as expected, and you want to avoid having to manually check that.

Eclipse provides a specific launch configuration, "Junit Plug-in Test", which executes Junit tests with a running Eclipse.

Implementing tests for the UI concepts might be tricky, since usually you will need to write code to set up Eclipse workbench infrastructures such as projects, files, and so on. Xtext provides some base classes for testing UI concepts which do most of the job for you, so that you can simply test specific features without having to worry about the setup steps.

> The examples in this section do not necessarily represent valuable tests; they should be seen as starting points for more complex tests of more complex DSLs.

Testing the content assist

To test the content assist, we use the base class `AbstractContentAssistTest`. We use the annotation `@RunWith` and `@InjectWith` as we did in the previous sections; however, for UI tests, we use the generated UI injector provider `EntitiesUiInjectorProvider` (note the "Ui" in the name):

```
@RunWith(typeof(XtextRunner))
@InjectWith(typeof(EntitiesUiInjectorProvider))
class EntitiesContentAssistTest extends
        AbstractContentAssistTest {
```

Then, in the test methods we can simply use the API of the `AbstractContentAssistTest` base class:

- `newBuilder` creates an object to test the content assist
- `append` appends some input text
- `assertText` declares the expected content assist proposals as strings (there are other assert methods, but this is the only one we will use)

Thus, our test class looks like this:

```
@RunWith(typeof(XtextRunner))
@InjectWith(typeof(EntitiesUiInjectorProvider))
class EntitiesContentAssistTest extends
        AbstractContentAssistTest {
  @Test
  def void testEmptyProgram() {
    newBuilder.assertText("entity")
  }

  @Test
  def void testSuperEntity() {
    newBuilder.append
      ("entity E extends ").assertText("E")
  }
  @Test
  def void testSuperEntity2() {
    newBuilder.append("entity A{} entity E extends ").
      assertText("A", "E")
  }

  @Test
  def void testAttributeTypes() {
    newBuilder.append("entity E { ").
      assertText
      ("E", "boolean", "int", "string", "}")
  }
}
```

Remember that you must run this class as a "Junit Plug-in Test".

> Make sure you have the bundle `org.eclipse.xtext.common.types.ui` as a dependency in the `MANIFEST.MF` file of the tests plug-in project.

Let us examine what the preceding tests do:

1. When the input is empty, we should get "entity" as the only proposal.

2. After an "extends", we should get as proposals the names of the available entities (with a single entity, we get its own name as the only proposal).

3. If there are two entities, we get both names as proposals.

4. After the entity opening curly bracket, we get as proposals all the types available, but also the closing curly bracket, since an entity can be defined without attributes.

Testing workbench integration

Sometimes you may want to test whether your DSL implementation "integrates" correctly with the Eclipse workbench. Usually, in these tests you also need a project in the running Eclipse.

In the following example we want to test whether a project with Entities DSL files builds correctly without errors.

Xtext provides the base class `AbstractWorkbenchTest` for testing Eclipse workbench related operations. In particular, this base class implements `@Before` and `@After` methods so that each `@Test` method is executed in a clean environment (that is, with an empty workbench) and the workbench is cleaned after each test (that is, all test projects are closed and deleted).

Besides that, Xtext provides the classes `IResourcesSetupUtil` and `JavaProjectSetupUtil` with many static utility methods to programmatically create workspace projects (we will import these static methods as static extension methods in the Xtend test class).

In our case, we want to create before each test, a Java workspace project and add the Xtext nature to that project (recall the dialog you get when you first add an entities file into a project); moreover, we also add `entities-gen` as a source folder. The ID of the Xtext nature is retrieved using the `XtextProjectHelper` class, which is contained in the bundle `org.eclipse.xtext.ui` (thus, you must add that as a dependency in the `.tests` plug-in project's `MANIFEST.MF` file).

We then write a reusable method which:

1. Creates an `entities` file in the workspace project with the specified contents.

2. Waits for Eclipse to build the project.

3. Checks whether there are error markers related to the `entities` file.

All these operations use the utility methods of the two preceding mentioned classes (the error markers are retrieved using the standard Eclipse API):

```
import static org.junit.Assert.*
import static extension org.eclipse.xtext.junit4.ui.util.
JavaProjectSetupUtil.*
import static extension org.eclipse.xtext.junit4.ui.util.
IResourcesSetupUtil.*

class EntitiesWorkbenchTest extends AbstractWorkbenchTest {

  val TEST_PROJECT = "mytestproject"

  @Before
  override void setUp() {
    super.setUp
    createJavaProjectWithXtextNature
  }

  def void createJavaProjectWithXtextNature() {
    createJavaProject(TEST_PROJECT) => [
      getProject().addNature
        (XtextProjectHelper::NATURE_ID)
      addSourceFolder("entities-gen")
    ]
  }

  def void checkEntityProgram(String contents,
      int expectedErrors) {
    val file = createFile(TEST_PROJECT +
      "/src/test.entities", contents)
    waitForAutoBuild();
    assertEquals(expectedErrors,
      file.findMarkers(EValidator::MARKER, true,
        IResource::DEPTH_INFINITE).size);
  }
```

Now we can write two test methods to check that for a valid entities file there is no error marker and that for a non-valid entities file we get an error marker:

```
@Test
def void testValidProgram() {
  checkEntityProgram("entity E {}", 0)
}

@Test
def void testNotValidProgram() {
  checkEntityProgram("foo", 1)
}
```

> You may have noticed that we did not use any injection mechanisms in the test; indeed, we only manipulate Eclipse concepts (such as projects, files, and so on) thus we do not need direct access to classes of our DSL implementation. Xtext is executing in the background while we programmatically create projects and files in the Eclipse test workbench.

Testing the editor

There are situations where you may want to perform specific test operations with the editor of your DSL. The test class in this case should derive from `AbstractEditorTest` (which in turn derives from `AbstractWorkbenchTest`). This base class provides utility methods to programmatically open an Eclipse editor on a file in the workbench (represented by `IFile`). The base class `AbstractEditorTest` requires us to implement the method `getEditorId` to return the ID of the editor of the DSL we want to test; the setup methods are similar to `EntitiesWorkbenchTest`, so we omit them here:

```
class EntitiesEditorTest extends AbstractEditorTest {

  ... as in EntitiesWorkbenchTest

  override protected getEditorId() {
    "org.example.entities.Entities"
  }

  def createTestFile(String contents) {
    createFile(TEST_PROJECT +
      "/src/test.entities", contents)
  }
```

```
@Test
def void testEntitiesEditor() {
  createTestFile("entity E {}").openEditor
}
```

The code also shows a first test which checks that we can open an editor for our DSL. Note that `openEditor` returns an instance of `XtextEditor`, which can be used to access (and possibly to modify) the editor contents.

When developing your DSL, you may want to add further features in the IDE which are not covered by the Xtext framework itself; for example, you may want to add context menus for the editor of your DSL that perform some actions on the contents of the editor and possibly change the contents. As we saw in *Chapter 6, Customizations*, Xtext already adds similar menus like "Format", or similar functionalities like quickfixes. In order to do that, you must learn the API of Eclipse editors and, in particular, of `XtextEditor` (which specializes the standard Eclipse text editor).

A nice way of learning a new API is to write **Learning Tests** (Beck 2002): these tests basically verify that the API works as expected. Thus, in the rest of this section we write some tests which allow us to learn how to use the `XtextEditor` API. Note that besides making us learn how to use this API, these tests will also guarantee that if that API changes in a non-backward compatible way in the future, we will realize that since these tests will fail.

The actual text edited by an Eclipse editor is stored in an `IDocument` instance and can be retrieved as a String using the method `get` (we have already used this in *Chapter 4, Validation*, when implementing quickfixes). This works also with `XtextEditor`, as shown by this test (which only checks that the editor's text is exactly the same as stored in the test file):

```
@Test
def void testEntitiesEditorContents() {
  "entity E {}".assertEquals(
    createTestFile("entity E {}").
      openEditor.document.get)
}
```

Usually, you will not need the string text, you will need the parsed EMF model. The class `XtextEditor` uses a specialization of `IDocument`, `IXtextDocument`, which provides access to the underlying parsed EMF model. However, in an Eclipse workbench, the access to the EMF model underlying an edited program must be performed in a "synchronized" way, since there are concurrent components running in different threads accessing such model (both UI components, like the editor itself, and non UI components, like the parser and the validator). Thus, for example, if you open an Xtext editor and you need the parsed EMF model, you will have to wait for the editor's text to be parsed. Similarly, if you want to modify the underlying model, you will have to wait until there are no other threads that are using it.

The EMF model of an Xtext editor's document can be accessed using the method `readOnly`; this method takes as an argument a lambda with a parameter of type `XtextResource` (a specialized EMF resource, as we saw when testing the formatter). We can retrieve the model from the passed `XtextResource`. When the body of the lambda is executed, there will be no other concurrent threads accessing the EMF resource. Thus, within the body of the lambda, the access to the model is guaranteed to be synchronized. This learning test reads the EMF model corresponding to the editor contents, retrieves the entity and checks that its name is as expected:

```
@Test
def void testEntitiesEditorContentsAsModel() {
  "E".assertEquals(
    createTestFile("entity E {}").
      openEditor.document.readOnly [
        // 'it' is an XtextResource
        contents.get(0) as Model
      ].entities.get(0).name
  )
}
```

The lambda passed to `readOnly` can return any type (Xtend will infer it). In the preceding test, the return type of the lambda is `Model`.

Similarly, we can access and modify the EMF model using the method `modify` and passing a lambda. The synchronization, in case of a modification, will also ensure that after the underlying model is modified, all the other components (like the parser and validator) will be notified.

The following test creates a test file with a single entity, `E`, and then modifies the EMF model by adding another entity, `Added`, which extends `E`; it then checks that the editor string contents have been updated accordingly:

```
@Test
def void testChangeContents() {
  val editor = createTestFile("entity E {}").openEditor

  editor.document.modify [
    val model = (contents.get(0) as Model)
    val currentEntity = model.entities.get(0)
    model.entities +=
    EntitiesFactory::eINSTANCE.createEntity => [
      name = "Added"
      superType = currentEntity
    ]
  ]
  '''
  entity E {}

  entity Added extends E {
  }'''.toString.assertEquals(editor.document.get)
}
```

Note that the resulting editor's text corresponding to the entity we added programmatically is formatted according to the formatter we implemented in *Chapter 6*, *Customizations*; in *Chapter 6*, *Customizations*, we also saw that the formatter is automatically triggered when selecting the quickfix provider based on semantic modification. Indeed, quickfix provider implementations are also automatically executed in a synchronous way.

Although the tests for the editor shown in this section are learning tests, they should give you an idea of how the Xtext editor API works.

Other UI testing frameworks

In the previous section, we saw that we are able to test UI concepts with the utility classes of Xtext.

These tests, however, do not test the actual operations that a user can perform in the IDE, like, for example, selecting a menu, right-clicking on an item and selecting a context menu, interacting with a dialog, and so on. The tests that confirm that a system does what the users are expecting it to are called **Functional Tests**.

If you need to test these UI features, you should consider using a functional UI testing framework. The most known in the Eclipse scenario are SWTBot (`http://www.eclipse.org/swtbot`) and Jubula (`http://www.eclipse.org/jubula`). Other frameworks (free, open source, or commercial) are briefly described at this URL: `http://wiki.eclipse.org/Eclipse/Testing`.

The treatment of these frameworks is out of the scope of this book.

Testing and modularity

One of the nice advantages of Test Driven Development is that it forces you to write modular code: it is not easy to test code that is not modular. Thus, either you give up on testing (an option which the author hopes you will never consider) or you decouple modules to easily test them. If you did not adopt this methodology from the beginning, remember that it is always possible to refactor the code to make it more modular and more testable. Thus, TDD and modular/decoupled design goes hand in hand and drives quality; well designed modular code is easier to test and well tested code has a known quality.

When evaluating whether to accept this programming methodology, you should also take into consideration that testing UI aspects is usually harder (as the examples in this chapter have shown). Thus, you should try to isolate the code that does not depend on a running Eclipse. Fortunately, in a DSL implementation this is easy.

Let us consider our quickfix provider we implemented in *Chapter 4*, *Validation*, which adds the missing referred entity; we show that here again for convenience:

```
@Fix(Diagnostic::LINKING_DIAGNOSTIC)
def void createMissingEntity(Issue issue,
                             IssueResolutionAcceptor acceptor) {
  acceptor.accept(issue,
    "Create missing entity",
    "Create missing entity",
    "Entity.gif",
    [ EObject element, IModificationContext context |
      val currentEntity =
        element.getContainerOfType(typeof(Entity))
      val model = currentEntity.eContainer as Model
      model.entities.add(model.entities.indexOf(currentEntity)+1,
        EntitiesFactory::eINSTANCE.createEntity() => [
          name =
            context.xtextDocument.get(issue.offset, issue.length)
          ]
        )
```

```
    ]
  );
}
```

Testing the quickfix provider is not straightforward. However, what we would really like to test here is that the desired entity is added in the right place in the model; this has nothing to do with the quickfix provider itself.

The manipulation of the EMF model does not need any IDE feature, thus we can isolate that in a utility class' static method, for example:

```
class EntitiesModelUtil {
  def static addEntityAfter(Entity entity,
         String nameOfEntityToAdd) {
    val model = entity.eContainer as Model
    EntitiesFactory::eINSTANCE.createEntity() => [
      name = nameOfEntityToAdd
      model.entities.add
        (model.entities.indexOf(entity)+1, it)
    ]
  }
}
```

Manipulating an EMF model is easy using the EMF API, but when the operations to perform on the model are complex, it is crucial to test them. Now that the code that modifies the EMF model is in a method that is independent from the UI, it is easy to test the insertion of an entity in the expected position:

```
class EntitiesModelUtilTest {
  val factory = EntitiesFactory::eINSTANCE

  @Test
  def void testAddEntityAfter() {
    val e1 = factory.createEntity => [name = "First"]
    val e2 = factory.createEntity => [name = "Second"]
    val model = factory.createModel => [
      entities += e1
      entities += e2
    ]

    EntitiesModelUtil::addEntityAfter(e1, "Added").
      assertNotNull
    3.assertEquals(model.entities.size)
```

```
        "First".assertEquals(model.entities.get(0).name)
        "Added".assertEquals(model.entities.get(1).name)
        "Second".assertEquals(model.entities.get(2).name)
    }
}
```

This checks that the new entity is inserted right after the desired existing entity (between `First` and `Second`).

Now we can also test that this code can be executed with an EMF model underlying an Xtext editor:

```
class EntitiesEditorTest extends AbstractEditorTest {
    … as in the preceding code
@Test
def void testAddEntity() {
    val editor = createTestFile(
    '''entity E1 {}

    entity E2 {}''').openEditor

    editor.document.modify [
        EntitiesModelUtil::addEntityAfter(
            (contents.get(0) as Model).entities.get(0),
            "Added")
    ]
    '''
    entity E1 {}

    entity Added {
    }

    entity E2 {}'''.toString.assertEquals(editor.document.get)
}
```

This shows that our method for adding an entity works also in the context of an Xtext editor and that its contents get updated consistently.

Now, we can refactor our quickfix provider method as follows:

```
@Fix(Diagnostic::LINKING_DIAGNOSTIC)
def void createMissingEntity(Issue issue,
                             IssueResolutionAcceptor acceptor) {
    acceptor.accept(issue,
        "Create missing entity", // label
        "Create missing entity", // description
```

```
        "Entity.gif", // icon
        [ EObject element, IModificationContext context |
          EntitiesModelUtil::addEntityAfter(
            element.getContainerOfType(typeof(Entity)),
            context.xtextDocument.get(issue.offset, issue.length)
          )
        ]
    );
}
```

We can safely assume that this quickfix provider will work, since it depends on concepts that we have already tested in isolation. We could retrieve the name of the entity to add in the wrong way, but that would be easier to spot in the preceding code than in the original implementation of the quickfix provider.

Clean code

Keeping your code "clean" (see Martin 2008, 2011) is important for the development of software (and this includes modularity, readability, and maintainability). Xtext provides many features to keep your DSL implementation clean and modular, thanks to its decomposition into many customizable aspects. Xtend extremely enhances the ability to write clean code thanks to its syntax and its advanced features such as lambda expressions and extension methods. In this book we will put much effort into writing clean code when implementing a DSL, in particular, we will try to write small methods and to factor common code into reusable methods.

Tests must be clean as well since they are part of the development cycle and they will have to be modified often. Remember that tests also provide documentation, thus they must be easily readable. In this chapter we tried to write small test methods by relying on reusable utility methods and classes. Note that writing small methods does not necessarily mean writing a few lines of code; for example, when we want to compare generated code, we need to embed the expectation in the test method. However, the number of expressions/statements is kept to the minimum, and the meaning of the test should be easily understood.

Summary

In this chapter we introduced unit testing for languages implemented with Xtext. Being able to test most of the DSL aspects without having to start an Eclipse environment really speeds up development.

Test Driven Development is an important programming methodology that helps you make your implementations more modular, more reliable, and resilient to changes of the libraries used by your code.

In the next chapter we will implement a DSL based on expressions; besides the apparent simplicity, parsing and checking expressions is not that trivial, since arithmetic and boolean expressions are inherently recursive, and dealing with recursion always requires some additional attention and effort. We will also implement a type system to check that expressions are well-typed.

8

An Expression Language

In this chapter we will implement a DSL for expressions, including arithmetic, boolean, and string expressions. We will do incrementally and in a test-driven way. Since expressions are by their own nature recursive, writing a grammar for this DSL requires some additional efforts, and this allows us to discover additional features of Xtext grammars.

You will also learn how to implement a type system for a DSL to check that expressions are correct with respect to types (for example, you cannot add an integer and a boolean). We will implement the type system so that it fits the Xtext framework and integrates correctly with the corresponding IDE tooling.

Finally, we will implement an interpreter for these expressions. We will use this interpreter to write a simple code generator that creates a text file with the evaluation of all the expressions of the input file, and also to show the evaluation of an expression in the editor.

The Expressions DSL

In the DSL we develop in this chapter, which we call **Expressions DSL**, we want to accept input programs consisting of expressions and variable declarations with an initialization expression. Expressions can refer to variables and can perform arithmetic operations, compare expressions, use logical connectors (`and` and `or`), and concatenate strings. We will use + both for representing arithmetic addition and for string concatenation; when used with strings, the + will also have to automatically convert integers and booleans occurring in such expressions into strings.

Here is an example of a program that we want to write with this DSL:

```
i = 0
j = (i > 0 && (1) < (i+1))
k = 1
```

```
j || true
"a" + (2 * (3 + 5)) // string concatenation
(12 / (3 * 2))
```

For example, `"a" + (2 * (3 + 5))` should evaluate to the string `"a16"`.

Creating the project

First of all, we will use the Xtext project wizard to create the projects for our DSL (following the same procedure explained in *Chapter 2*, *Creating Your First Xtext Language*).

Start Eclipse and perform the following steps:

1. Navigate to **File** | **New** | **Project...**, in the dialog navigate to the **Xtext** category and click on **Xtext Project**.

2. In the next dialog provide the following values:

   ° **Project name**: `org.example.expressions`
   ° **Name**: `org.example.expressions.Expressions`
   ° **Extensions**: `expressions`
   ° Uncheck the option **Create SDK feature project**

The wizard will create three projects and it will open the file `Expressions.xtext`, which is the grammar definition.

Digression on Xtext grammar rules

Before writing the Xtext grammar for the Expressions DSL, it is important to spend some time to understand how the rules in an Xtext grammar and the corresponding EMF model generated for the AST are related.

From the previous chapters we know that Xtext, while parsing the input program, will create a Java object corresponding to the used grammar rule. Let us go back to our Entities DSL example and consider the rule:

```
Entity:
  'entity' name = ID ('extends' superType=[Entity])? '{'
    attributes += Attribute*
  '}' ;
```

When the parser uses this rule it will create an instance of the `Entity` class (that class has been generated by Xtext during the MWE2 workflow). However, the actual creation of such an instance will take place on the first assignment to a feature of the rule; in this example, no object will be created when the input only contains `entity`; the object will be created as soon as a name is specified, for example, when the input contains `entity A`. This happens because such an ID is assigned to the feature `name` in the rule. This is reflected in the outline view, as shown in the following screenshots:
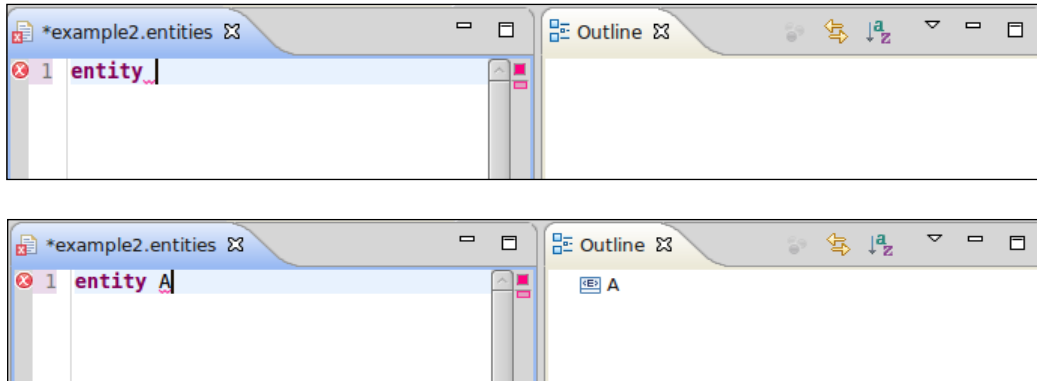
This also means that the created `Entity` object is not "complete" at this stage, that is, when only a part of the rule has been applied. That is why when writing parts of the DSL implementation, for example, the validator, the UI customizations, and so on, you must always take into consideration that the model you are working on may have some features set to `null`.

The actual creation of the object of the AST can be made explicit by the programmer using the notation {type name} inside the rule; for example:

```
Entity:
  'entity' {Entity} name = ID ('extends' superType=[Entity])? '{'
    attributes += Attribute*
  '}' ;
```

If you change the rule as shown, then an `Entity` object will be created as soon as the `entity` keyword has been parsed, even if there has not been a feature assignment.

In the examples we have seen so far, the type of the object corresponds to the rule name; however, the type to instantiate can be specified using `returns` in the rule definition:

```
A returns B:
  ... rule definition ...
;
```

In this example, the parser will create an instance of B. A is simply the name of the rule, and there will be no generated A class. Indeed, the shape of the rule definitions we have used so far is just a shortcut for:

```
A returns A:
  ... rule definition ...
;
```

That is, if no `returns` is specified, the type corresponds to the name of the rule.

Moreover, the `returns` statement and the explicit `{type name}` notation can be combined:

```
A returns B:
  ... {C} ... rule definition ...
;
```

In this example, the parser will create an instance of C (and C is generated as a subclass of B). However, the object returned by the rule will have type B. Also in this case, there will be no generated A class.
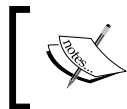
> When defining a cross-reference in the grammar, the name enclosed in square brackets refers to a type, not to a rule's name, unless they are the same.

When writing the grammar for the Expressions DSL, we will use these features.

The grammar for the Expressions DSL

The DSL that we want to implement in this chapter should allow us to write lines containing either a variable consisting of an identifier and an initialization expression (the angle brackets denote non-terminal symbols):

```
<ID> = <Expression>
```

or an expression by itself:

```
<Expression>
```

If we write something like this

```
ExpressionsModel:
  variables += Variable*
  expressions += Expression*
;
```

We will not be able to write a program where variables and expressions can be defined in any order; we could only write variables first and then expressions.

To achieve the desired flexibility, we introduce an abstract class for both variable declarations and expressions; then, our model will consist of a (possibly empty) sequence of such abstract elements.

For the moment, we consider a very simple kind of expression: integer constants. These are the first rules (we skip the initial declaration parts of the grammar):

```
ExpressionsModel:
  elements += AbstractElement*;

AbstractElement:
  Variable | Expression ;

Variable:
  name=ID '=' expression=Expression;

Expression:
  value=INT;
```

The generated EMF classes for `Variable` and `Expression` will be subclasses of `AbstractElement`.

We are ready to write the first tests for this grammar. This chapter assumes that you fully understood the previous chapter about testing; thus, the code for testing we show here should be clear:

```
@RunWith(typeof(XtextRunner))
@InjectWith(typeof(ExpressionsInjectorProvider))
class ExpressionsParserTest {

  @Inject extension ParseHelper<ExpressionsModel>
  @Inject extension ValidationTestHelper

  @Test def void testSimpleExpression() {
    '''10'''.parse.assertNoErrors
  }

  @Test def void testVariableExpression() {
    '''i = 10'''.parse.assertNoErrors
  }
}
```

These methods test that both variable declarations and expressions can be parsed without errors.

We now continue adding rules; for example, we want to parse string and boolean constants besides integer constants.

We could write a single rule for all these constant expressions:

```
Expression:
(intvalue=INT)|
(stringvalue=STRING)|
(boolvalue=('true'|'false'));
```

But this would not be good. It is generally not a good idea to have constructs that result in a single class that represents multiple language elements, since later when we are performing validation and other operations we cannot differentiate on class alone and instead have to inspect the corresponding fields (which, as you may recall, may not always be initialized due to partial parsing).

It is much better to write a separate rule for each element as shown in the following code snippet, and this will lead to the generation of separate classes:

```
Expression:
  IntConstant | StringConstant | BoolConstant;

IntConstant: value=INT;
StringConstant: value=STRING;
BoolConstant: value=('true'|'false');
```

Note that, although `IntConstant`, `StringConstant`, and `BoolConstant` will all be subclasses of `Expression`, the field `value` will not be part of the `Expression` superclass; for `IntConstant` the field `value` will be of type integer, while for the other two it will be of type string; thus it cannot be made common.

At this point you must run the MWE2 workflow and make sure that the previous tests still run successfully; then you should add additional tests for parsing a string constant and a boolean constant (this is left as an exercise). We can write the aforementioned rules in a more compact form, using the notation `{type name}` that we introduced in the previous section, *Digression on Xtext grammar rules*:

```
Expression:
  {IntConstant} value=INT |
  {StringConstant} value=STRING |
  {BoolConstant} value=('true'|'false');
```

Again, run the workflow and make sure tests still pass.

We add a rule which accepts a reference to an existing variable as follows:

```
Expression:
  {IntConstant} value=INT |
  {StringConstant} value=STRING |
  {BoolConstant} value=('true'|'false') |
  {VariableRef} variable=[Variable];
```

To test this last modification, we need an input with a variable declaration and an expression that refers to that variable:

```
@Test def void testVariableReference() {
  val e =
  '''
  i = 10
  i
  '''.parse
  e.assertNoErrors;

  (e.elements.get(1) as VariableRef).variable.
    assertSame(e.elements.get(0))
}
```

Note that we also test that the variable reference actually corresponds to the declared variable; assertSame comes from org.junit.Assert, whose static methods have been imported as extension methods.

Left recursive grammars

When moving on to a more complex expression, such as addition, we need to write a recursive rule since the left and right parts of an addition are expressions themselves. It would be natural to express such a rule as follows:

```
Expression:
  ... as above
  {Plus} left=Expression '+' right=Expression;
```

However, this results in an error from the Xtext editor as shown in the following screenshot:
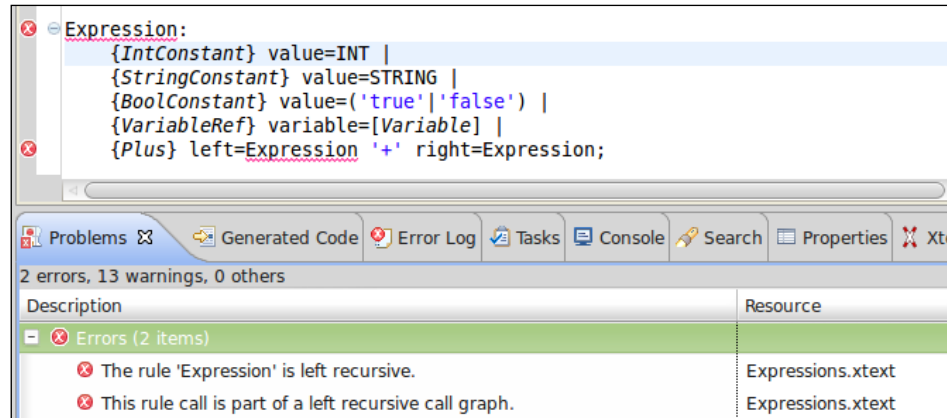
Xtext uses a parser algorithm that is suitable for use for interactive editing due to its better handling of error recovery. Unfortunately this parser algorithm does not deal with left recursive rules. A rule is **left recursive** when the first symbol of the rule is non-terminal and refers to the rule itself. The preceding rule for addition is indeed left recursive and is rejected by Xtext.

> Xtext generates an ANTLR parser (Parr 2007), which relies on an LL(*) algorithm; we will not go into details about parsing algorithms; we refer the interested reader to Aho et al. 2007. Such parsers have nice advantages concerning debugging and error recovery, which are essential in an IDE to provide a better feedback to the programmer. However, such parsers cannot deal with left recursive grammars.

The good news is that we can solve this problem, the bad news is that we have to modify the grammar to remove the left recursion, using a transformation referred to as **left-factoring**.

The parser generated by ANTLR cannot handle left recursion since it relies on a top-down strategy. Bottom-up parsers do not have this problem, but they would require to handle **operator precedence** (which determines which sub-expressions should be evaluated first in a given expression) and **associativity** (which determines how operators of the same precedence are grouped in the absence of parentheses). As we will see in this section, left-factoring will allow us to implicitly define operator precedence and associativity.

You may be tempted to remove the left recursion by simply adding a token before the recursion, for instance, parentheses:

```
Expression:
  ... as above
  {Plus} '(' left=Expression '+' right=Expression ')';
```

But clearly this is not a solution; our DSL would accept addition expressions only if enclosed in parentheses. We will add parentheses to the grammar later, for the purpose of allowing expression grouping as a means to influence precedence –– for example, `10/(5+5) => 1` instead of `10/5+5 => 7`.

Instead, we remove the left recursion using a standard technique: we introduce a rule for expressions which are **atomic** and we state that an addition consists of a left part, which is an atomic expression, and an optional right part, which is recursively an expression (note that the right recursion does not disturb the ANTLR parser):

```
Expression:
  {Plus} left=Atomic ('+' right=Expression)?;

Atomic returns Expression:
  {IntConstant} value=INT |
  {StringConstant} value=STRING |
  {BoolConstant} value=('true'|'false') |
  {VariableRef} variable=[Variable];
```

Remember that the rule name is `Atomic`, but objects in the AST created by this rule will be of type `IntConstant`, `StringConstant`, and so on, according to the alternative used (indeed, no `Atomic` class will be generated from the preceding rule). Statically, these objects will be considered of type `Expression`, and thus the `left` and `right` fields in the `Plus` class will be of type `Expression`.

The preceding solution still has a major drawback, that is, additional useless nodes will be created in the AST. For example, consider an atomic expression such as a variable reference; when the atomic expression is parsed using the preceding rule, the AST will consist of a `Plus` object where the `VariableRef` object is stored in the `left` feature:
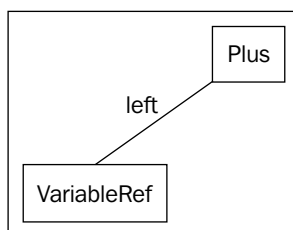
This indirection tends to be quite disturbing. For instance, the previous test method `testVariableReference` will now fail due to a `ClassCastException` and must be modified as follows:

```
@Test def void testVariableReference() {
  ...

  ((e.elements.get(1) as Plus).left as VariableRef).variable.
    assertSame(e.elements.get(0))
}
```

What we need is a way of telling the parser to:

- Try to parse an expression using the `Atomic` rule
- Search for an optional + followed by another expression
- If the optional part is not found then the expression is the element parsed with the `Atomic` rule
- Otherwise, instantiate a `Plus` object where `left` is the previously parsed expression with `Atomic` and `right` is the expression parsed after the +

All these operations can be expressed in an Xtext grammar as follows:

```
Expression:
  Atomic ({Plus.left=current} '+' right=Expression)? ;
```

The part {Plus.left=current} is a **tree rewrite action,** and it does what we want: if the part (...)? can be parsed, then the resulting tree will consist of a `Plus` object where `left` is assigned the subtree previously parsed by the `Atomic` rule and `right` is assigned the subtree recursively parsed by the `Expression` rule.

Now, the test method `testVariableReference` can go back to its original form, since parsing an atomic expression does not result in an additional `Plus` object.

Associativity

Associativity instructs the parser how to build the AST when there are several infix operators with the same precedence in an expression. It will also influence the order in which elements of the AST should be processed in an interpreter or compiler.

What happens if we try to parse something like 10 + 5 + 1? The parsing rule invokes the rule for `Expression` recursively; the rule is right recursive, and thus we expect the preceding expression to be parsed in a **right-associative** way, that is, 10 + 5 + 1 will be parsed as 10 + (5 + 1). In fact, the optional part (...)? can be used only once; thus, the only way to parse 10 + 5 + 1 is to parse 10 with the `Atomic` rule and the rest with the optional part.
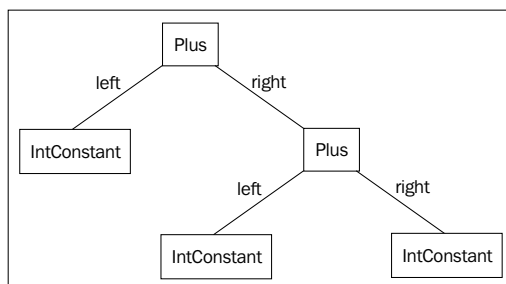
If, on the contrary, we write the rule as follows:

```
Expression:
  Atomic ({Plus.left=current} '+' right=Atomic)* ;
```

We will get **left associativity**, that is, `10 + 5 + 1` will be parsed as `(10 + 5) + 1`. In fact, the optional part `(...)*` can be used many times; thus, the only way to parse `10 + 5 + 1` is to apply that part twice (note that `right=Atomic` and not `right=Expression` as in the previous section).
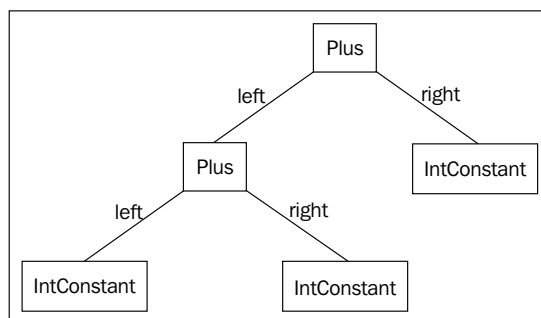
It is important to check that the associativity of the parsed expressions is as expected. A simple way to check the result of associativity is to generate a string representation of the AST where non-atomic expressions are enclosed in parentheses.

```
def stringRepr(Expression e) {
  switch (e) {
    Plus:
    '''(«e.left.stringRepr» + «e.right.stringRepr»)'''
    IntConstant: '''«e.value»'''
    StringConstant: '''«e.value»'''
    BoolConstant: '''«e.value»'''
    VariableRef: '''«e.variable.name»'''
  }.toString
}
```

Then we write a method `assertRepr(input,expected)` that checks that the associativity of the input corresponds to the expected representation:

```
def assertRepr(CharSequence input, CharSequence expected) {
  input.parse => [
    assertNoErrors;
    expected.assertEquals(
      (elements.last as Expression).stringRepr
    )
  ]
}
```

We will use this method in the rest of the section to test the associativity of expressions. For instance, for testing the associativity of an addition, we write:

```
@Test def void testPlus() {
  '''10 + 5 + 1 + 2'''.assertRepr("(((10 + 5) + 1) + 2)")
}
```

We add a rule for parsing expressions in parentheses:

```
Atomic returns Expression:
  '(' Expression ')' |
  {IntConstant} value=INT |
  ... as above
```

Note that the rule for parentheses does not perform any assignment to features; thus, given the parsed text `(exp)`, the AST will contain a node for `exp`, not for `(exp)`. This can be verified by this test:

```
@Test def void testParenthesis() {
  "(10)".parse.elements.get(0) as IntConstant
}
```

Although parentheses will not be part of the AST, they will influence the structure of the AST:

```
@Test def void testPlusWithParenthesis() {
  "( 10 + 5 ) + ( 1 + 2 )".assertRepr("((10 + 5) + (1 + 2))")
}
```

For the sum operator, left associativity and right associativity are equivalent (indeed, sum is an **associative operation**); the same holds for multiplication (with the possible exception of overflows or loss of precision that depends on the order of evaluation). This does not hold for subtraction and division. In fact, how we parse and then evaluate such operations influences the result: `(3 – 2) – 1` is different from `3 – (2 – 1)`.

In these cases, you can disable associativity by using this pattern for writing the grammar rule:

```
Expression:
  Atomic ({Operation.left=current} '-' right=Atomic)? ;
```

In fact the `?` operator (instead of `*`) does not allow to parse an `Atomic` on the right more than once. This way, the user will be forced to explicitly use grouping expressions for the operations with no associativity when there are more than two operands.

The alternative solution is to choose an associativity strategy for the parser and implement the evaluator accordingly. Typically arithmetic operations are parsed and evaluated in a left-associative way (this holds in Java for instance), and we prefer this solution since users will get no surprises with this default behavior. Of course, parentheses can always be used for grouping. We modify the grammar for dealing with subtractions; we want to use a dedicated class for a subtraction expression, for example, `Minus`. We then modify the rule for `Expression` as follows:

```
Expression:
  Atomic (
    ({Plus.left=current} '+' | {Minus.left=current} '-')
    right=Atomic
  )* ;
```

Note that the tree rewrite action inside the rule is selected according to the parsed operator. Though we are not doing that in this section (due to lack of space), you should write a test with a subtraction operation; of course, you must also update the `stringRepr` utility method for handling the case for `Minus`.

Precedence

We can write the case for addition and subtraction in the same rule because they have the same arithmetic operator precedence. If we add a rule for multiplication and division we must handle their precedence with respect to addition and subtraction.

To define the precedence we must write the rule for the operator with less precedence in terms of the rule for the operator with higher precedence. This means that in the grammar, the rules for operators with less precedence are defined first. Since multiplication and division have higher precedence than addition and subtraction, we modify the grammar as follows:

```
Expression: PlusOrMinus;

PlusOrMinus returns Expression:
```

```
    MulOrDiv (
      ({Plus.left=current} '+' | {Minus.left=current} '-')
      right=MulOrDiv
    )* ;

  MulOrDiv returns Expression:
    Atomic (
      ({MulOrDiv.left=current} op=('*'|'/'))
      right=Atomic
    )* ;
```

In the preceding rules, we use `returns` to specify the type of the created objects; thus, the features `left` and `right` in the corresponding generated Java classes will be of type `Expression`.

We added a main rule for `Expression` which delegates to the first rule to start parsing the expression; remember that this first rule (at the moment `PlusOrMinus`) concerns the operators with lowest precedence. There will be no class called `PlusOrMinus` since only objects of class `Plus` and `Minus` will be created by this rule. On the contrary in the rule `MulOrDiv`, we create objects of class `MulOrDiv`. In this rule we also chose another strategy: we have a single object type, `MulOrDiv`, both for multiplication and division expressions. After parsing, we can tell between the two using the operator which is saved in the feature `op`. Whether to have a different class for each expression operator or group several expression operators into one single class is up to the developer; both strategies have their advantages and drawbacks, as we will see in the next sections.

We now test the precedence of these new expressions:

```
@Test
def void testPlusMulPrecedence() {
  "10 + 5 * 2 - 5 / 1".assertRepr("((10 + (5 * 2)) - (5 / 1))")
}

def stringRepr(Expression e) {
  switch (e) {
    Plus:
    '''(«e.left.stringRepr» + «e.right.stringRepr»)'''
    Minus:
    '''(«e.left.stringRepr» - «e.right.stringRepr»)'''
    MulOrDiv:
    '''(«e.left.stringRepr» «e.op» «e.right.stringRepr»)'''
... as before
```

Now we add boolean expressions and comparison expressions to the DSL; again, we have to deal with their precedence, which is as follows, starting from the ones with less precedence:

1. **boolean or** (operator ||)
2. **boolean and** (operator &&)
3. **equality** and **dis-equality** (operators == and !=, respectively)
4. **comparisons** (operators <, <=, >, and >=)
5. **addition** and **subtraction**
6. **multiplication** and **division**

Following the same strategy for writing the grammar rules, we end up with the following expression grammar:

```
Expression: Or;

Or returns Expression:
  And ({Or.left=current} "||" right=And)* ;

And returns Expression:
  Equality ({And.left=current} "&&" right=Equality)* ;

Equality returns Expression:
  Comparison (
    {Equality.left=current} op=("=="|"!=")
    right=Comparison
  )* ;

Comparison returns Expression:
  PlusOrMinus (
    {Comparison.left=current} op=(">="|"<="|">"|"<")
    right=PlusOrMinus
  )* ;

... as before
```

When writing tests for these new expressions, we need to test for the correct precedence for the new operators both in isolation and in combination with other expressions (remember to update the stringRepr method to handle the new classes). We leave these tests as an exercise; they can be found in the sources of the examples. While adding rules for new expressions, we also added test methods in our parser test class, and we run all these tests each time: this will not only ensure that the new rules work correctly, but also that they do not break existing rules.

As a final step, we add a rule for boolean negation (operator "!"); this operator has the highest precedence among all the operators seen so far. Therefore, we can simply add a case in the `Atomic` rule:

```
Atomic returns Expression:
  '(' Expression ')' |
  {Not} "!" expression=Atomic |
  ... as before
```

We can now write a test with a complex expression and check that the parsing takes place correctly:

```
@Test def void testPrecedences() {
  "!true||false&&1>(1/3+5*2)".
    assertRepr
      ("((!true) || (false && (1 > ((1 / 3) + (5 * 2)))))")
}
```

Now might be a good time to refactor the rule `Atomic`, since it includes cases that are not effective atomic elements. We introduce the rule `Primary` for expressions with the highest priority that need some kind of evaluation; the rule `MulOrDiv` is refactored accordingly:

```
MulOrDiv returns Expression:
  Primary (
    {MulOrDiv.left=current} op=('*'|'/')
    right=Primary
  )*
;

Primary returns Expression:
  '(' Expression ')' |
  {Not} "!" expression=Primary |
  Atomic
;

Atomic returns Expression:
  {IntConstant} value=INT |
  {StringConstant} value=STRING |
  {BoolConstant} value=('true'|'false') |
  {VariableRef} variable=[Variable]
;
```

The complete grammar

We sum up the section by showing the complete grammar of the Expressions DSL:

```
grammar org.example.expressions.Expressions with
  org.eclipse.xtext.common.Terminals

generate expressions
  "http://www.example.org/expressions/Expressions"

ExpressionsModel:
  elements += AbstractElement*;

AbstractElement:
  Variable | Expression ;

Variable:
  name=ID '=' expression=Expression;

Expression: Or;

Or returns Expression:
  And ({Or.left=current} "||" right=And)*
;

And returns Expression:
  Equality ({And.left=current} "&&" right=Equality)*
;

Equality returns Expression:
  Comparison (
    {Equality.left=current} op=("=="|"!=")
    right=Comparison
  )*
;

Comparison returns Expression:
  PlusOrMinus (
    {Comparison.left=current} op=(">="|"<="|">"|"<")
    right=PlusOrMinus
  )*
;

PlusOrMinus returns Expression:
  MulOrDiv (
```

```
      ({Plus.left=current} '+' | {Minus.left=current} '-')
      right=MulOrDiv
   )*
;

MulOrDiv returns Expression:
   Primary (
      {MulOrDiv.left=current} op=('*'|'/')
      right=Primary
   )*
;

Primary returns Expression:
   '(' Expression ')' |
   {Not} "!" expression=Primary |
   Atomic
;

Atomic returns Expression:
   {IntConstant} value=INT |
   {StringConstant} value=STRING |
   {BoolConstant} value=('true'|'false') |
   {VariableRef} variable=[Variable]
;
```

Forward references

You should know by now that parsing is only the first stage when implementing a DSL and that it cannot detect all the errors from the programs. We need to implement additional checks in a validator.

One important thing we need to check in our Expressions DSL is that an expression (including the initialization expression of a variable) does not refer to a variable defined after the very expression. Using an identifier before its declaration is usually called a **forward reference**.

Therefore, this program should not be considered valid:

```
i = j + 1
j = 0
```

since the initialization expression of i refers to j, which is defined after. Of course, this is a design choice: since we want to interpret the expressions, it makes sense to interpret them in the order they are defined.

This strategy also avoids possible mutual dependency problems:

```
i = j + 1
j = i + 1
```

A variable which is initialized referring to itself is a special case of the preceding:

```
i = i + 1
```

We want to avoid this because our interpreter would enter an endless loop when evaluating expressions; this choice is also consistent with the design choice that variables, once initialized, cannot be further assigned.
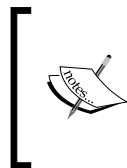
> Restricting the visibility of references can also be implemented with a custom `ScopeProvider`; the subject of **Scoping** will be detailed in *Chapter 10, Scoping*. However, **visibility** and **validity** are not necessarily the same mechanism. Therefore, it also makes sense to implement this checking in the validator.

We write a `@Check` method in the `ExpressionsValidator` class to report such problems.

To summarize, given a variable reference inside an expression, we need to:

- Get the list of all the variables defined before the containing expression
- Check that in the list there is a variable with the name of the reference

Note that we need to do that only if the variable is correctly bound; that is, if the cross-reference has already been resolved by Xtext, otherwise an error has already been reported.

The harder part is getting the list of all the variables defined before the containing expression. First of all, given an expression, we must get the `AbstractElement` containing such expression. For example, in an isolated expression, say `10`, such element is the expression itself; if we have `k = (10 + 5) < (j * 2)`, and we want to validate the variable reference `j`, then the element we need is the whole `k = (10 + 5) < (j * 2)`. This functionality only deals with model traversing, and we isolate it into a utility class, `ExpressionsModelUtil` (we have already seen this technique in *Chapter 7, Testing,* section *Testing and Modularity*). We will also reuse this utility class when implementing other parts of this DSL.

We implement the static method `variablesDefinedBefore(AbstractElement e)`. In this method, we get the containing root model object using the method `org.eclipse.xtext.EcoreUtil2.getContainerOfType`. From the root, we get the list of all the elements. We then need to find the element containing `e`, that is, either `e` itself, or the element which contains `e`. We use the static method `org.eclipse.emf.ecore.util.EcoreUtil.isAncestor` to get that. Then, we get the sublist of variables. This is implemented in Xtend as follows:

```
import static extension org.eclipse.emf.ecore.util.EcoreUtil.*
import static extension org.eclipse.xtext.EcoreUtil2.*

class ExpressionsModelUtil {
  def static variablesDefinedBefore(AbstractElement e) {
    val allElements =
      e.getContainerOfType(typeof(ExpressionsModel)).elements
    val containingElement =
      allElements.findFirst[isAncestor(it, e)]
    allElements.subList(0,
      allElements.indexOf(containingElement)).
        typeSelect(typeof(Variable))
  }
}
```

> This algorithm is potentially slow when dealing with large models and complex logic since it always searches from the top element down. Depending on your DSL, you might want to implement other strategies, such as bottom-up search, creating an index upfront, and so on. Once you have a bunch of tests for the simple implementation, you can experiment with optimizations and make sure that the tests still succeed.

We can now test this class in a separate JUnit test class (we show only a snippet) to make sure it does what we expect:

```
@Test def void variablesBeforeVariable() {
  '''
  true    // (0)
  i = 0    // (1)
  i + 10    // (2)
  j = i    // (3)
  i + j    // (4)
  '''.parse => [
    assertVariablesDefinedBefore(0, "")
    assertVariablesDefinedBefore(1, "")
    assertVariablesDefinedBefore(2, "i")
```

```
      assertVariablesDefinedBefore(3, "i")
      assertVariablesDefinedBefore(4, "i,j")
  ]
}

def void assertVariablesDefinedBefore(ExpressionsModel model,
      int elemIndex, CharSequence expectedVars) {
  expectedVars.assertEquals(
    model.elements.get(elemIndex).variablesDefinedBefore.
    map[name].join(",")
  )
}
```

Writing the `@Check` method in the validator is easy now:

```
import static extension org.example.expressions.typing.
ExpressionsModelUtil.*

class ExpressionsValidator extends AbstractExpressionsValidator {
  public static val FORWARD_REFERENCE =
    "org.example.expressions.ForwardReference";

  @Check
  def void checkForwardReference(VariableRef varRef) {
    val variable = varRef.getVariable()
    if (variable != null &&
        !varRef.variablesDefinedBefore.contains(variable))
      error("variable forward reference not allowed: '"
        + variable.name + "'",
        ExpressionsPackage::eINSTANCE.variableRef_Variable,
        FORWARD_REFERENCE, variable.name)
  }
}
```

It is also easy to test it (we show only a snippet):

```
@Test
def void testForwardReferenceInExpression() {
  '''i = 1 j+i j = 10'''.parse => [
      assertError(ExpressionsPackage::eINSTANCE.variableRef,
        ExpressionsValidator::FORWARD_REFERENCE,
        "variable forward reference not allowed: 'j'"
      )
      // check that it is the only error
      1.assertEquals(validate.size)
    ]
}
```

Note that in this test we also make sure that there is only one error concerning `j` and not `i`, which is correctly used after its declaration. We test this using the list of issues returned by `ValidationTestHelper.validate`. We issue an error in case of a forward reference, but we did not customize the way Xtext resolves references, thus the user of the Expressions DSL can still jump to the actual declaration of the variable, also in case of a forward reference error. This is considered a good thing in the IDE; for example, in JDT if you try to access a private member of a different class, you get an error, but you can still jump to the declaration of that member.

At the same time, though, the content assist of our DSL will also propose variables defined after the context where we are writing in the editor; this should be avoided since it is misleading. We can fix this by customizing the content assist. Xtext already generated a stub class in the UI plug-in project, for customizing this; in this example, it is `org.example.expressions.ui.contentassist.ExpressionsProposalProvider` in the `src` folder. Also this class relies on a method signature convention as shown in the following code snippet:

```
public void complete{RuleName}_{FeatureName} (
  EObject element, Assignment assignment,
  ContentAssistContext context,
  ICompletionProposalAcceptor acceptor)
public void complete_{RuleName} (
  EObject element, RuleCall ruleCall,
  ContentAssistContext context,
  ICompletionProposalAcceptor acceptor)
```

In the signatures, `RuleName` is the name of the rule in the grammar and `FeatureName` is the name of the feature (with the first letter capitalized) assigned in that rule. The idea is to use the first method signature to customize the proposals for a specific feature of that rule and the second one for customizing the proposals for the rule itself. In our case, we want to customize the proposals for the `variable` feature in the `Atomic` rule (that is, the rule which parses a `VariableRef`); thus we define a method called `completeAtomic_Variable`. The stub class extends a generated class in the `src-gen` folder, `AbstractExpressionsProposalProvider` that implements all of these `complete` methods. You can inspect the base class to get the signature right. You should also use the `@Override` annotation so that you get an error if the name of the rule or of the feature changes in the grammar. Most of the time, you will only use the first parameter, which is the `EObject` object representing the object corresponding to the rule being used and the acceptor to which you will pass your custom proposals.

In the method `completeAtomic_Variable`, we get the variables defined before the passed `EObject` and create a proposal for each variable:

```
override completeAtomic_Variable(EObject elem,
  Assignment assignment,
```

```
      ContentAssistContext context,
      ICompletionProposalAcceptor acceptor) {
      if (!(elem instanceof AbstractElement))
        return; // no proposal

      (elem as AbstractElement).variablesDefinedBefore.forEach[
        variable |
        acceptor.accept(createCompletionProposal
          (variable.name,
          variable.name + " - Variable", null,
          context));
      ]
    }
```

Proposals are created using the `createCompletionProposal` method; you need to pass the string which will be inserted in the editor, the string shown in the content assist menu, a default image, and the context you received as a parameter. Each proposal must be passed to the acceptor.

> Remember that for a given offset in the input program file, there can exist several possible grammar elements. Xtext will dispatch to the method declarations for any valid element, and thus many `complete` methods may be called.

In the previous chapter, we showed you how to test the content assist; we will now test our custom implementation (we only show the relevant parts):

```
    @Test
    def void testVariableReference() {
      newBuilder.append("i = 10 1+").
      assertText('!', '"Value"', '(', '+', '1', 'false', 'i', 'true')
    }

    @Test
    def void testForwardVariableReference() {
      newBuilder.append("k= 0 j=1 1+  i = 10 ").
        assertTextAtCursorPosition("+", 1,
        '!', '"Value"', '(', '+', '1', 'false', 'j', 'k', 'true')
    }
```

In the first method, we verify that we did not remove variable proposals that are valid; in the second one, we verify that only the variables that are defined before the current context are proposed. In particular, in the second test, we used an assert method passing the character in the input to set the (virtual) cursor at and an additional offset: we ask for the proposals right after the first + in the input string. We test that j and k are proposed, but not i.

Typing expressions

In the Expressions DSL, types are not written explicitly by the programmer. However, due to the simple nature of our expressions, we can easily deduce the type of an expression by looking at its shape. In this DSL we have a fixed set of types: string, integer, and boolean. The mechanism of deducing a type for an expression is usually called **type computation** or **type inference**.

The base cases for type computation in the Expressions DSL are constants; trivially, an integer constant has type integer, a string constant has type string, and a boolean constant has type boolean.

As for composed expressions, besides computing a type, we must also check that its sub-expressions are correct with respect to types. This mechanism is usually called **type checking**. For example, consider the expression `!e`, where `e` is a generic expression. We can say that it has type boolean, provided that, recursively, the sub-expression `e` has type boolean; otherwise, the whole expression is not **well-typed**.

All the type mechanisms are part of the **type system** of the language. The type system depends on the semantics, that is, the meaning that we want to give to the elements of the DSL. For the Expressions DSL we design a type system that reflects the natural treatment of arithmetic and boolean expressions, in particular:

- If in a `Plus` expression one of the sub-expressions has type string, the whole expression is considered to have type string; if they have both type integer, then the whole expression has type integer; two boolean expressions cannot be added
- `Equality` can only act on sub-expressions with the same type
- `Comparison` can only act on sub-expressions with the same type, but not on booleans

Of course, a type system should also be consistent with code generation or interpretation; this is typically formally proved, but this is out of the scope of the book (we refer the interested reader to Hindley 1987, Cardelli 1996, and Pierce 2002). For instance, in our DSL, we allow expressions of the shape `1 + "a"` and `"a" + true`: we consider these expressions to have type string, since we use `+` also for string concatenation and implicit string conversion. If in a `Plus` expression the two sub-expressions have both type integer, the whole expression will have type integer, since that will be considered as the arithmetic addition. During interpretation, we must interpret such expressions accordingly.

Typically, type computation and type checking are implemented together and, as just seen, they are recursive. In general, the type of an expression depends on the type of the sub-expressions; the whole expression is not well-typed if any of its sub-expressions are not well-typed. Here are some examples: `j * true` is not well-typed since multiplication is defined only on integers; `true == "abc"` is not well-typed since we can only compare by equality expressions of the same type, `true < false` is not well-typed since comparison operators do not make sense on booleans, and so on.

When implementing a type system in an Xtext DSL, we must take into consideration a few aspects: Xtext automatically validates each object in the AST, not only on the first level elements. For instance, if in our validator we have these two `@Check` methods:

```
@Check
public void checkType(And and)

@Check
public void checkType(Not not)
```

and the input expression is `!a && !b` then Xtext will automatically call the second method on the `Not` objects corresponding to `!a` and `!b` and the first method on the object `And` corresponding to the whole expression. Therefore, it makes no sense to perform recursive invocations ourselves inside the validator's methods. If an expression contains some sub-expressions which are not well-typed, it should be considered not well-typed itself; however, does it make sense to mark the whole expression with an error? For example, consider this expression:

```
(1 + 10) < (2 * (3 + "a"))
```

If we mark the whole expression as not well-typed, we would provide useless information to the programmer. The same holds true if we generate additional errors for the sub-expressions `(2 * (3 + "a"))` and `(3 + "a")`. Indeed, the useful information is that `(3 + "a")` has type string while an integer type was expected by the multiplication expression.

Due to all the aforementioned reasons, we adopt the following strategy:

- Type computation is performed without recurring on sub-expressions unless required in some cases; for example, a `MulOrDiv` object has type integer independently of its sub-expressions. This is implemented by the class `ExpressionsTypeProvider`.

- In the validator, we have a `@Check` method for each kind of expression, excluding the constant expressions, which are implicitly well-typed; each method checks that the types of the sub-expressions, obtained by using `ExpressionsTypeProvider`, are as expected by that specific expression. For example, for a `MulOrDiv`, we check that its sub-expressions have both type integer, otherwise, we issue an error on the sub-expression that does not have type integer.

As we will see, this strategy avoids checking the same object with the validator several times, since the type computation is delegated to `ExpressionsTypeProvider`, which is not recursive. It will also allow the validator to generate meaningful error markers only on the problematic sub-expressions.

Type provider

Since we do not have types in the grammar of the Expressions DSL, we need a way of representing them. Since the types for this DSL are simple, we just need an interface for types, for example, `ExpressionsType`, and a class implementing it for each type, for example, `StringType`, `IntType`, and `BoolType`. These classes implement a `toString` method for convenience, but they do not contain any other information.
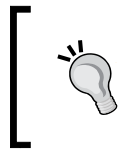
> We write the classes for types and for the type provider in the new Java sub-package `typing`. If you want to make its classes visible outside the main plug-in project, you should add this package to the list of exported packages in the **Runtime** tab of the `MANIFEST.MF` editor.

In the type provider, we define a static field for each type. Using singletons will allow us to simply compare a computed type with such static instances.

```
class ExpressionsTypeProvider {
  public static val stringType = new StringType
  public static val intType = new IntType
  public static val boolType = new BoolType
```

We now write a method, `typeFor`, which, given an `Expression`, returns an `ExpressionsType` object. We use the `dispatch` methods for special cases and `switch` for simple cases. For expressions whose type can be computed directly, we write:

```
def dispatch ExpressionsType typeFor(Expression e) {
  switch (e) {
    StringConstant: stringType
    IntConstant: intType
    BoolConstant: boolType
```

```
      Not: boolType
      MulOrDiv: intType
      Minus: intType
      Comparison: boolType
      Equality: boolType
      And: boolType
      Or: boolType
  }
}
```

We now write a test class for our type provider with several test methods
(we only show some of them):

```
import static extension org.junit.Assert.*

@RunWith(typeof(XtextRunner))
@InjectWith(typeof(ExpressionsInjectorProvider))
class ExpressionsTypeProviderTest {
  @Inject extension ParseHelper<ExpressionsModel>
  @Inject extension ExpressionsTypeProvider

  @Test def void intConstant() { "10".assertIntType }
  @Test def void stringConstant() { "'foo'".assertStringType }
...
  @Test def void notExp() { "!true".assertBoolType }

  @Test def void multiExp() { "1 * 2".assertIntType }
  @Test def void divExp() { "1 / 2".assertIntType }
...
  def assertStringType(CharSequence input) {
    input.assertType(ExpressionsTypeProvider::stringType)
  }
...
  def assertType(CharSequence input,
    ExpressionsType expectedType) {
      expectedType.assertSame
        (input.parse.elements.last.typeFor)
  }
}
```

We wrote the method `assertType` that does most of the work: it parses the passed
input and computes the type of the last element of the program.

Then, we can move on to more elaborate type computations, which we implement in a `dispatch` method for better readability.

```
def dispatch ExpressionsType typeFor(Plus e) {
  val leftType = e.left?.typeFor
  val rightType = e.right?.typeFor
  if (leftType == stringType || rightType == stringType)
    stringType
  else
    intType
}
```

For `Plus`, we need to compute the type of sub-expressions, since if one of them has type string, the whole expression is considered to be a string concatenation (with implicit conversion to string); thus, we give it a string type. Otherwise, it is considered to be the arithmetic sum and we give it type integer. In this type system, this is the only case where type computation depends on the types of sub-expressions.

Remember that the type provider can also be used on a non-complete model, and thus we use the null-safe operator `?.`, described in *Chapter 3*, *The Xtend Programming Language*.

We can now test this case for the type provider:

```
@Test def void numericPlus() { "1 + 2".assertIntType }
@Test def void stringPlus() { "'a' + 'b'".assertStringType }
@Test def void numAndStringPlus() { "'a' + 2".assertStringType }
@Test def void numAndStringPlus2() { "2 + 'a'".assertStringType }
@Test def void boolAndStringPlus(){"'a' + true".assertStringType }
@Test def void boolAndStringPlus2(){"false+'a'".assertStringType }
```

Also, the case for variable reference requires some more work:

```
def dispatch ExpressionsType typeFor(VariableRef varRef) {
  if (varRef.variable == null ||
    !(varRef.variablesDefinedBefore.contains(varRef.variable)))
      return null
  else
      return varRef.variable.expression?.typeFor
}
```

We must check that the reference concerns a variable defined before the current expression, otherwise we might enter an infinite loop; to this aim, we reuse the `ExpressionsModelUtil` class whose static methods are here imported as extension methods. We must also consider the case when the reference is not resolved. In both cases, we simply return `null`, and we know that an error has already been reported (by our own validator in the former case and by the default validator for cross-references in the latter case). Otherwise, the type of a variable reference is the type of the referred variable, which, in turn, is the type of its initialization expression. We must check that the initialization expression is not `null`, in case of an incomplete program. The test for this case of the type provider is as follows:

```
@Test def void varRef() { "i = 0  i".assertIntType }
```

> Remember that you should follow the **Test Driven Development** strategy illustrated in *Chapter 7*, *Testing*; first, write the implementation for a single kind of expression, write a test for that case then execute it; then, proceed with the implementation for another kind of expression, write a test for that case and run all the tests, and so on. As an exercise, you should try to re-implement everything seen in this section from scratch yourself following this methodology.

Validator

We are ready to write the `@Check` methods in the `ExpressionsValidator`.

> It is possible that some of the tests for the parser you have previously written fail now due to the validator methods and due to the fact that when you wrote those test expressions, you were not considering types. This is a standard situation in test driven development, and there is nothing to worry about. You can either modify the test expressions in the parser tests so that they are correct also with respect to types, or simply remove the call to `assertNoErrors`. In the latter case, the tests still make sense: when checking the associativity and precedence by traversing the AST, if the parsing failed due to a syntax error, the AST would be incomplete and the tests would fail with null pointer exception.

In the existing validator, we inject an instance of `ExpressionsTypeProvider` and we write some reusable methods which perform the actual checks. Thanks to these methods, we will be able to write the `@Check` methods in a very compact form.

```
class ExpressionsValidator extends
  AbstractExpressionsValidator {

  public static val WRONG_TYPE =
```

```
      "org.example.expressions.WrongType";

  @Inject extesion ExpressionsTypeProvider

  def private checkExpectedBoolean(Expression exp,
                                    EReference reference) {
    checkExpectedType(exp,
      ExpressionsTypeProvider::boolType, reference)
  }

  def private checkExpectedInt(Expression exp,
                                EReference reference) {
    checkExpectedType(exp,
      ExpressionsTypeProvider::intType, reference)
  }

  def private checkExpectedType(Expression exp,
      ExpressionsType expectedType, EReference reference) {
    val actualType = getTypeAndCheckNotNull(exp, reference)
    if (actualType != expectedType)
      error("expected " + expectedType +
        " type, but was " + actualType,
        reference, WRONG_TYPE)
  }

  def private ExpressionsType getTypeAndCheckNotNull(
        Expression exp, EReference reference) {
    var type = exp?.typeFor
    if (type == null)
      error("null type", reference, WRONG_TYPE)
    return type;

  }...
```

> Although we present the reusable methods first, the actual strategy
> we followed when implementing the type checking in the validator is
> to first write some @Check methods, and then refactor the common
> parts. Before refactoring, we wrote some tests; after refactoring, we
> execute the tests to verify that refactoring did not break anything.

It should be straightforward to understand what the aforementioned methods do.
The methods are parameterized over the EMF feature to use when generating an error;
remember that this feature will be used to generate the error marker appropriately.

Let us see some `@Check` methods:

```
@Check def checkType(Not not) {
  checkExpectedBoolean(not.expression,
    ExpressionsPackage$Literals::NOT__EXPRESSION)
}

@Check def checkType(And and) {
  checkExpectedBoolean(and.left,
    ExpressionsPackage$Literals::AND__LEFT)
  checkExpectedBoolean(and.right,
    ExpressionsPackage$Literals::AND__RIGHT)
}
```

For `Not`, `And`, and `Or`, we check that the sub-expressions have type boolean and we pass the EMF features corresponding to the sub-expressions. (The case for `Or` is similar to the case of `And`, and it is therefore not shown).

Following the same approach, it is easy to check that the sub-expressions of `Minus` and `MultiOrDiv` both have integer types (we leave this as an exercise, but you can look at the sources of the example).

For an `Equality` expression, we must check that the two sub-expressions have the same type. This holds true also for a `Comparison` expression, but in this case, we also check that the sub-expressions do not have type boolean, since in our DSL, we do not want to compare two boolean values. The implementation of these `@Check` methods are as follows, using two additional reusable methods:

```
@Check def checkType(Equality equality) {
  val leftType = getTypeAndCheckNotNull(equality.left,
    ExpressionsPackage$Literals::EQUALITY__LEFT)
  val rightType = getTypeAndCheckNotNull(equality.right,
    ExpressionsPackage$Literals::EQUALITY__RIGHT)
  checkExpectedSame(leftType, rightType)
}

@Check def checkType(Comparison comparison) {
  val leftType = getTypeAndCheckNotNull(comparison.left,
    ExpressionsPackage$Literals::COMPARISON__LEFT)
  val rightType = getTypeAndCheckNotNull(comparison.right,
    ExpressionsPackage$Literals::COMPARISON__RIGHT)
  checkExpectedSame(leftType, rightType)
  checkNotBoolean(leftType,
    ExpressionsPackage$Literals::COMPARISON__LEFT)
  checkNotBoolean(rightType,
```

```
      ExpressionsPackage$Literals::COMPARISON__RIGHT)
}


def private checkExpectedSame(ExpressionsType left,
                             ExpressionsType right) {
  if (right != null && left != null && right != left) {
    error("expected the same type, but was "+left+", "+right,
      ExpressionsPackage$Literals::EQUALITY.getEIDAttribute(),
      WRONG_TYPE)
  }
}


def private checkNotBoolean(ExpressionsType type,
                           EReference reference) {
  if (type == ExpressionsTypeProvider::boolType) {
    error("cannot be boolean", reference, WRONG_TYPE)
  }
}
```

The final check concerns the Plus expression; according to our type system, if one of the two sub-expressions has type string, everything is fine, and therefore all these combinations are accepted as valid: string+string, int+int, string+boolean, and string+int (and the corresponding specular cases). We cannot add two boolean expressions or an integer and a boolean. Therefore, when one of the two sub-expressions has type integer or when they both have a type different from string, we must check that they do not have type boolean:

```
@Check def checkType(Plus plus) {
  val leftType = getTypeAndCheckNotNull(plus.left,
    ExpressionsPackage$Literals::PLUS__LEFT)
  val rightType = getTypeAndCheckNotNull(plus.right,
    ExpressionsPackage$Literals::PLUS__RIGHT)
  if (leftType == ExpressionsTypeProvider::intType
      || rightType == ExpressionsTypeProvider::intType
      || (leftType != ExpressionsTypeProvider::stringType &&
          rightType != ExpressionsTypeProvider::stringType)) {
    checkNotBoolean(leftType,
      ExpressionsPackage$Literals::PLUS__LEFT)
    checkNotBoolean(rightType,
      ExpressionsPackage$Literals::PLUS__RIGHT)
  }
}
```

Of course, while writing these methods, we also wrote test methods in the
`ExpressionsValidatorTest` class. Due to lack of space, we are not showing these
tests, and instead we refer you to the source code of the Expressions DSL.

Let's try the editor and look at the error markers as shown in the following screenshot:
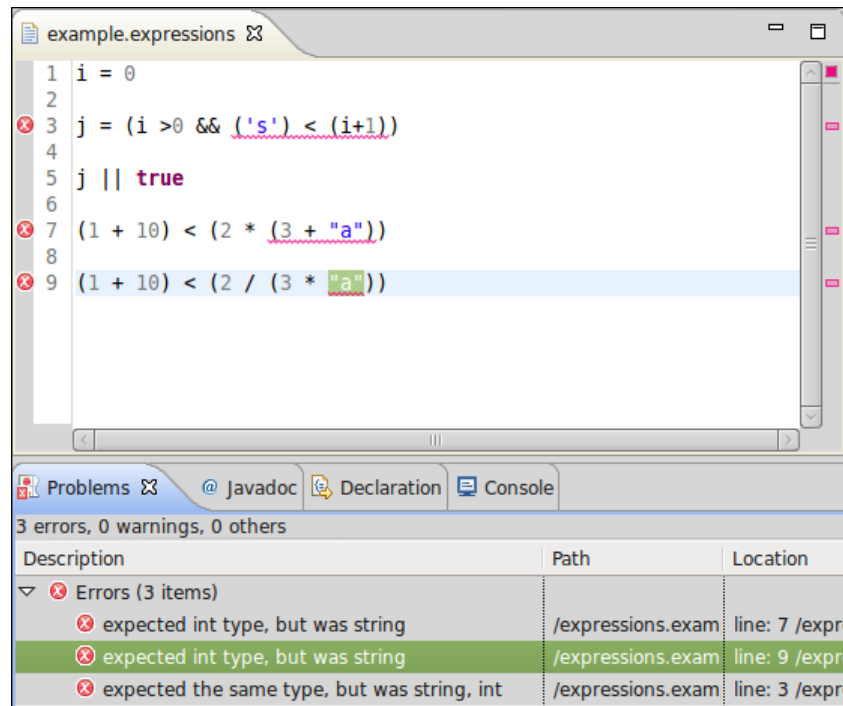
The error markers are placed only on the sub-expression that is not well-typed;
it is clear where the problem inside the whole expression is. If we did not follow
the preceding strategy for computing and checking types, in a program with some
not well-typed expressions, most of the lines would be red, and this would not
help. With our implementation, the expression `j || true` does not have error
markers, although the initialization expression of `j` contains an error; our type
provider is able to deduce that `j` has type boolean anyway. Formally, also `j`,
and in turn `j || true`, are not well-typed; however, marking `j || true` with an
error would only generate confusion.

Writing an interpreter

We will now write an interpreter for our Expressions DSL. The idea is that this interpreter, given an `AbstractElement`, returns a Java object which represents the evaluation of that element. Of course, we want the object with the result of the evaluation to be of the correct Java type; that is, if we evaluate a boolean expression, the corresponding object should be a Java boolean object.

Such an interpreter will be recursive, since to evaluate an expression, we must first evaluate its sub-expressions and then compute the result.

When implementing the interpreter we make the assumption that the passed `AbstractElement` is valid. Therefore, we will not check for null sub-expressions; we will assume that all variable references are resolved and we will assume that all the sub-expressions are well-typed; for example, if we evaluate an `And` expression, we assume that the objects resulting from the evaluation of its sub-expressions are Java `Boolean` objects.

For constants, the implementation of the evaluation is straightforward:

```
class ExpressionsInterpreter {

  def dispatch Object interpret(Expression e) {
    switch (e) {
      IntConstant: e.value
      BoolConstant: Boolean::parseBoolean(e.value)
      StringConstant: e.value
```

Note that the feature `value` for an `IntConstant` object is of Java type `int` and for a `StringConstant` object, it is of Java type `String`, and thus we do not need any conversion. For a `BoolConstant` object the feature `value` is also of Java type `String`, and thus we perform an explicit conversion using the static method of the Java class `Boolean`.

As usual, we immediately start to test our interpreter, and the actual assertions are all delegated to a reusable method:

```
class ExpressionsInterpreterTest {
  @Inject extension ParseHelper<ExpressionsModel>
  @Inject extension ValidationTestHelper
  @Inject extension ExpressionsInterpreter

  @Test def void intConstant() { "1".assertInterpret(1) }
  @Test def void boolConstant() { "true".assertInterpret(true) }
  @Test def void stringConstant() {"'abc'".assertInterpret("abc") }

  def assertInterpret(CharSequence input, Object expected) {
    input.parse => [
```

```
        assertNoErrors
        expected.assertEquals(elements.last.interpret)
    ]
}...
```

Note that, in order to correctly test the interpreter, we check that there are no errors in the input (since that is the assumption of the interpreter itself) and we compare the actual objects, not their string representation. This way, we are sure that the object returned by the interpreter is of the expected Java type.

Then, we write a case for each expression. We recursively evaluate the sub-expressions, and then apply the appropriate Xtend operator to the result of the evaluation of the sub-expressions. For example, for And:

```
switch (e) {
...
And: {
    (e.left.interpret as Boolean) && (e.right.interpret as Boolean)
}
```

Note that the method `interpret` returns an `Object`, and thus we need to cast the result of the invocation on sub-expressions to the right Java type. We do not perform an `instanceof` check because, as hinted previously, the interpreter assumes that the input is well-typed.

With the same strategy, we implement all the other cases. We show here only the most interesting ones. For `MulOrDiv`, we will need to check the actual operator, stored in the feature `op`:

```
switch (e) {
...
 MulOrDiv: {
  val left = e.left.interpret as Integer
  val right = e.right.interpret as Integer
  if (e.op == '*')
    left * right
  else
    left / right
}
```

For `Plus`, we need to perform some additional operations: since we use + both as the arithmetic sum and as string concatenation, we must know the type of the sub-expressions. We use the type provider and write:

```
class ExpressionsInterpreter {
  @Inject extension ExpressionsTypeProvider
```

```
def dispatch Object interpret(Expression e) {
  switch (e) {
  ...
  Plus: {
    if (e.left.typeFor.isString || e.right.typeFor.isString)
      e.left.interpret.toString + e.right.interpret.toString
    else
      (e.left.interpret as Integer) +
          (e.right.interpret as Integer)
  }...
```

The method `isString` is a utility method that we added to
`ExpressionsTypeProvider` to avoid doing the comparison with string types.

Finally, we deal with the case of variable and variable reference:

```
def dispatch Object interpret(Expression e) {
  switch (e) {
    ...
    VariableRef: e.variable.interpret
    ...
}

def dispatch Object interpret(Variable v) {
  v.expression.interpret
}
```

Using the interpreter

Xtext allows us to customize all UI aspects, as we saw in *Chapter 6*, *Customizations*.
We can provide a custom implementation of **text hovering** (that is, the pop-up
window that comes up when we hover for some time on a specific editor region)
so that it shows the type of the expression and its evaluation. We refer to the Xtext
documentation for the details of the customization of text hovering; here, we only
show our implementation (note that we create a multi-line string using HTML syntax):

```
class ExpressionsEObjectHoverProvider extends
    DefaultEObjectHoverProvider {
  @Inject extension ExpressionsTypeProvider
  @Inject extension ExpressionsInterpreter
  override getHoverInfoAsHtml(EObject o) {
    if (o instanceof AbstractElement && o.programHasNoError) {
      val elem = o as AbstractElement
      return '''
      <p>
      type  : <b>«elem.typeFor.toString»</b> <br>
      value : <b>«elem.interpret.toString»</b>
      </p>
      '''
    } else
      return super.getHoverInfoAsHtml(o)
  }

  def programHasNoError(EObject o) {
    Diagnostician::INSTANCE.validate(o.rootContainer).
      children.empty
  }
}
```

Remember that our interpreter is based on the assumption that it is invoked only on an EMF model that contains no error. We invoke our validator programmatically using the EMF API, that is, the `Diagnostician` class; we must validate the entire AST; thus, we retrieve the root of the EMF model using the method `EcoreUtil.getRootContainer` and check that the list of validation issues is empty. We need to write an explicit bind method for our custom implementation of text hovering in the `ExpressionsUiModule`:

```
public Class<? extends IEObjectHoverProvider>
    bindIEObjectHoverProvider() {
  return ExpressionsEObjectHoverProvider.class;
}
```

In the following screenshot, we can see our implementation when we place the mouse over the `*` operator of the expression `2 * (3 + 5)`: the pop-up window shows the type and the evaluation of the corresponding multiplication expression:
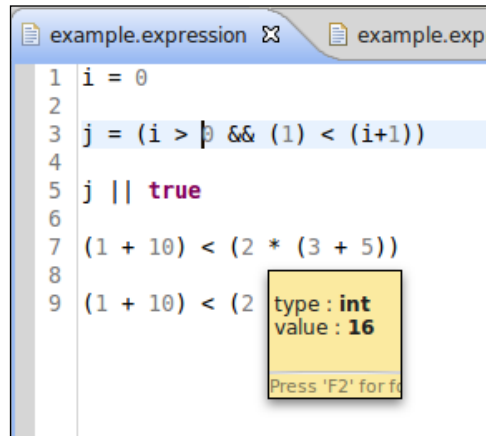
Finally, we can write a code generator which creates a text file (by default, it will be created in the directory `src-gen`):

```
import static extension
  org.eclipse.xtext.nodemodel.util.NodeModelUtils.*

class ExpressionsGenerator implements IGenerator {
  @Inject extension ExpressionsInterpreter

  override void doGenerate(Resource resource,
                           IFileSystemAccess fsa) {
    resource.allContents.toIterable.
      filter(typeof(ExpressionsModel)).forEach[
        fsa.generateFile
          ('''«resource.URI.lastSegment».evaluated''',
            interpretExpressions)
      ]
  }

  def interpretExpressions(ExpressionsModel model) {
    model.elements.map[
      '''«getNode.getTokenText» ~> «interpret»'''
    ].join("\n")
  }
}
```

Differently from the code generator we saw in *Chapter 5*, *Code Generation,* here we generate a single text file for each input file (an input file is represented by an EMF `Resource`); the name of the output file is the same as the input file (retrieved by taking the last part of the URI of the resource), with an additional `evaluated` file extension.

Instead of simply generating the result of the evaluation in the output file, we also generate the original expression. This can be retrieved using the Xtext class `NodeModelUtils`. The static utility methods of this class allow us to easily access the elements of the **node model** corresponding to the elements of the AST model. (Recall from *Chapter 6*, *Customizations* that the node model carries the syntactical information, for example, offsets and spaces of the textual input.) The method `NodeModelUtils.getNode(EObject)` returns the node in the node model corresponding to the passed `EObject`. From the node of the node model, we retrieve the original text in the program corresponding to the `EObject`.

An example input file and the corresponding generated text file are shown in the following screenshot:

```
example.expressions ⊠
1  i = 0
2  j = (i > 0 && (1) < (i+1))
3  k = 1
4  j || true
5
6  'a' + (2 * (3 + 5))
7
8  (12 / (3 * 2))
```

```
example.expressions.evaluated ⊠
1 i = 0 ~> 0
2 j = (i > 0 && (1) < (i+1)) ~> false
3 k = 1 ~> 1
4 j || true ~> true
5 'a' + (2 * (3 + 5)) ~> a16
6 12 / (3 * 2) ~> 2
```

Summary

In this chapter, we implemented a DSL for expressions; this allowed us to explore some techniques for dealing with recursive grammar rule definitions in Xtext grammars and some simple type checking. We also showed how to write an interpreter for an Xtext DSL.

In the next chapter we will develop a small object-oriented DSL. We will use this DSL to show some advanced type checking techniques that deal with object-oriented features such as inheritance and subtyping (type conformance).

9

Type Checking

In this chapter we will develop a small object-oriented DSL, which can be seen as a smaller version of Java that we call **SmallJava**. We will use this DSL to show some type checking techniques that deal with object-oriented features such as inheritance and subtyping (type conformance). This will also allow us to learn other features of Xtext grammars and to see some good practices in Xtext DSL implementations.

SmallJava

The language we develop in this chapter is a simplified version of Java, called SmallJava. This language does not aim at being useful in practice and cannot be used to write real programs such as Java. However, SmallJava contains enough language features that will allow us to explore advanced type checking techniques that can also be reused for other DSLs which have OOP mechanisms such as inheritance and subtyping.

The implementation we see in this chapter will not be complete, since some features of this language, such as correct member access, will be implemented in the next chapter when we introduce the mechanism of local and global scoping. In a Java-like language type checking and scoping are tightly connected and complement each other; for the sake of readability, we will split typing and scoping into two separate chapters.

We will not write a code generator or an interpreter for SmallJava: we are more interested in statically checking SmallJava programs rather than executing them. However, a code generator or an interpreter could easily be implemented reusing the same techniques illustrated in *Chapter 5*, *Code Generation* and *Chapter 8*, *An Expression Language* respectively.

Let us stress that implementing the whole Java language and, in particular, its complete type system, would not be feasible in this book. The Java type system is complex due to advanced features such as inner classes, static methods, method overloading, and generics. Furthermore, all these concepts require a solid background on type theory, which is out of the scope of this book. Instead we concentrate on a small subset of Java features, which, as hinted previously, are common also to other object-oriented languages. If your DSL needs to access Java types and to be interoperable with Java, you may want to consider using Xbase, briefly described in *Chapter 12*, *Xbase*. If your DSL does not have to interact with Java, the concepts described in this chapter can be reused and adapted to fit your DSL.

Creating the project

First of all, we will use the Xtext project wizard to create the projects for our DSL (following the same procedure that we saw in the previous chapters).

Start Eclipse and:

1. Navigate to **File** | **New** | **Project...**; in the dialog, navigate to the **Xtext** category and click on **Xtext Project**.
2. In the next dialog, fill in the details for the following fields:

    ° **Project name**: `org.example.smalljava`
    ° **Name**: `org.example.smalljava.SmallJava`
    ° **Extensions**: `smalljava`
    ° Uncheck the option **Create SDK feature project**

The wizard will create three projects into your workspace and it will open the `SmallJava.xtext` file that is the grammar definition.

SmallJava grammar

Before starting to develop this language, we sketch the simplifications we will adopt:

- Classes have no explicit constructors
- There is no cast expression
- Arithmetic and boolean expressions are not implemented
- Basic types (such as `int`, `boolean`, and so on) and void methods are not considered (methods must always return something)
- There is no method overloading

- Member access must always be prefixed with the object (even if it is `this`)
- `super` is not supported (but it will be implemented in the next chapter)
- The `new` instance expression does not take arguments (since there are only implicit default constructors)

Basically, the features that we are interested in and that will allow us to have a case study for type checking and scoping (next chapter) are class inheritance, field and method definitions, and blocks of statements with local variable definitions.

Rules for declarations

The rules in the grammar are prefixed with SJ to avoid confusion with the classes and terms in Java that they mimic.

The first rule is straightforward it states that a SmallJava program is a possibly empty sequence of classes:

```
SJProgram:
  classes+=SJClass*;

SJClass: 'class' name=ID ('extends' superclass=[SJClass])? '{'
  members += SJMember*
  '}' ;

SJMember:
  SJField | SJMethod ;

SJField:
  type=[SJClass] name=ID ';' ;

SJMethod:
  type=[SJClass] name=ID
  '(' (params+=SJParameter (',' params+=SJParameter)*)? ')'
  body=SJMethodBody ;
```

Each class can have a superclass (that is, a reference to another SmallJava class) and a possibly empty sequence of members. An SJMember object can be either an SJField object or an SJMethod object; note that since both fields and methods have a type and a name feature, these two features will end up in their common base class SJMember.

The body of a method is a sequence of `SJStatement` (defined later) enclosed in curly brackets:

```
SJMethodBody:
  '{' statements += SJStatement* '}';
```

If we define the rule for the method body as in the preceding code snippet, we get a warning:

```
The rule 'SJMethodBody' may be consumed without object
instantiation. Add an action to ensure object creation, for example,
'{SJMethodBody}'.
```

In fact, the only assignment is to the feature `statements`, which is based on `SJStatement*`; if no statement is parsed, the rule will be valid, but the feature will not be assigned and no object will be instantiated (see the *Digression on Xtext grammar rules* section, *Chapter 8*, *An Expression Language* for an explanation of how object instantiation and feature assignment in a rule are connected). As suggested by the warning, we add an action to ensure object creation:

```
SJMethodBody:
  {SJMethodBody} '{' statements += SJStatement* '}';
```

Rules for statements and syntactic predicates

These are the rules for statements:

```
SJStatement:
  SJVariableDeclaration |
  SJReturn |
  SJExpression ';' |
  SJIfStatement
;

SJReturn:
  'return' expression=SJExpression ';'
;

SJVariableDeclaration:
  type=[SJClass] name=ID '=' expression=SJExpression ';'
;

SJIfStatement:
  'if' '(' expression=SJExpression ')' thenBlock=SJIfBlock
  (=>'else' elseBlock=SJIfBlock)?
;
```

```
SJIfBlock:
  statements += SJStatement
  | '{' statements += SJStatement+ '}' ;
```

The blocks for an `if` statement can also be specified without curly brackets; in this case, a single statement can be specified.

The rule for the `if` statement shows another important feature of Xtext grammars: **syntactic predicates** represented by the symbol `=>`. These are useful to solve ambiguities in a grammar; we will use the `if` statement as an example to describe such situations. If we write the rule for the `if` statement as follows:

```
SJIfStatement:
  'if' '(' expression=SJExpression ')' thenBlock=SJIfBlock
  ('else' elseBlock=SJIfBlock)?
;
```

during the MWE2 workflow we will get this warning in the console:

```
warning(200): Decision can match input such as "'else'" using multiple
alternatives: 1, 2
As a result, alternative(s) 2 were disabled for that input
```

This issue is also known as the **Dangling Else Problem**; if you consider this nested `if` statement:

```
if (...)
  if (...)
    statement
  else
    statement
```

The parser would not know to which `if` statement the `else` belongs to (remember that spaces and indentation are ignored by the parser); it could belong either to the outer `if` or to the inner one, and this leads to an ambiguity.

> Never ignore warnings issued by the MWE2 workflow concerning the generated ANTLR parser: each warning will surely be a source of problems if not solved, even when ambiguities are automatically resolved by the parser itself using a default strategy. Make sure each ambiguity is resolved the way you want.

By using the syntactic predicate, i.e., the `=>` before the `'else'` keyword, we remove such ambiguity by directing the parser: we tell the parser that if it finds the `'else'` keyword, then it should not try to search for other ways of parsing the statement, and the `else` will belong to the inner `if` (which is the typical behavior of languages with an `if` statement).
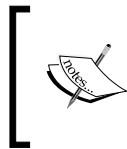
> Another way of solving ambiguities is to enable **backtracking** for the ANTLR parser generator. However, this is strongly discouraged, and it is only a source of further problems not easily detectable; thus we will not consider this option in this book.

The block for the method body and the one for the `if` statement are similar:

```
SJMethodBody:
  {SJMethodBody} '{' statements += SJStatement* '}';


SJIfBlock:
  statements += SJStatement
  | '{' statements += SJStatement+ '}' ;
```

In particular, they both have the `statements` feature. Therefore we add a rule in the grammar that introduces a common base class (with the `statements` feature):

```
SJBlock: SJMethodBody | SJIfBlock ;
```

Note that the rule name is grayed in the Xtext grammar editor: this highlights the fact that this rule is never called by the parser; however, the Java class `SJBlock` will be generated as the superclass for `SJMethodBody` and `SJIfBlock`. In particular, since the feature `statements` is common to both classes, this field will be in the class `SJBlock` so that we can treat both a method body and the blocks of an `if` statement in the same way. This will be useful when dealing with variable declarations in nested program blocks in the next chapter.

Similarly, the rule `SJParameter` and the rule `SJVariableDeclaration` have something in common: the `type` and `name` features. Therefore, we introduce a rule for forcing the creation of a common superclass:

```
SJSymbol: SJVariableDeclaration | SJParameter ;
```

This will give us a common supertype for referring to a local variable and a parameter from within a method body, as we will see later in this chapter.

Rules for expressions

As we saw in the rule `SJStatement`, a statement can also be an expression terminated by `;`. In fact, as in Java, a method invocation can be both a standalone statement and an expression, that is, the right-hand side of an assignment or the argument for another method invocation.

Also an assignment statement can itself be an expression; for instance, in Java you can write:

```
a = b = c ;
```

with the meaning that `c` is assigned both to `b` and to `a` (formally, first `c` is assigned to `b` and then `b` is assigned to `a`). Moreover, both the left-hand and right-hand side of an assignment can be expressions.

For all of the aforementioned reasons, and from what you learned from *Chapter 8*, *An Expression Language*, you should know that trying to write the rule for the assignment like this:

```
SJExpression: SJAssignment;

SJAssignment:
  left=SJExpression '=' right=SJExpression
;
```

would make the grammar left recursive.

A similar problem is experienced when writing the rule for a member selection expression, that is, for selecting a field or a method on an object (the most typical expression in an object-oriented language). In fact, in order to be able to write selection expressions like:

```
a.b().c.d()
```

the left-hand side of a member selection (which is usually called the **receiver** of the selection) must be an expression.

Therefore, we cannot write the rule for selection as follows, since it would be left recursive:

```
// This is invalid
SJExpression: SJAssignment | SJSelectionExpression;

SJSelectionExpression:
  receiver=SJExpression '.' member=[SJMember]
  ...
;
```

Therefore, we need to left factor the grammar using the technique introduced in *Chapter 8*, *An Expression Language*.

To keep the presentation simple, we will not deal with arithmetic and boolean expressions in the grammar. We will concentrate only on assignments, member selections, and some additional terminal expressions. All the other expressions can be easily added taking the Expressions DSL of *Chapter 8*, *An Expression Language* as inspiration.

We must write the rule for the expression with lower precedence first (that is, the rule for assignment):

```
SJExpression: SJAssignment;

SJAssignment returns SJExpression:
  SJSelectionExpression
  ({SJAssignment.left=current} '=' right=SJExpression)?
;
```

The left factoring technique should be familiar by now. Just note that an assignment expression must be right-associative (refer to *Chapter 8*, *An Expression Language* for associativity). That is why we wrote the right recursive part as:

```
right=SJExpression)?       Right associativity
```

instead of

```
right=SJSelectionExpression)*    Left associativity
```

Therefore, a nested assignment expression like

```
a = b = c
```

will be parsed as

```
a = (b = c)
```

as required by our DSL.

The rule for member selection is as follows:

```
SJSelectionExpression returns SJExpression:
  SJTerminalExpression
  (
    {SJMemberSelection.receiver=current} '.'
    member=[SJMember]
    (methodinvocation?='('
```

```
      (args+=SJExpression (',' args+=SJExpression)*)? ')'
    )?
  )* ;
```

Remember that while the name of the rule is `SJSelectionExpression`, the rule will instantiate an `SJMemberSelection` object.

Note that this rule deals both with field selection and method invocation; for this reason, the part that deals with method invocation can have optional parentheses and arguments. We keep track of the presence of parentheses using the boolean feature `methodinvocation`; this will allow us to distinguish between an `SJMemberSelection` object that represents a field selection from one that represents a method invocation.

You may also be tempted to distinguish between a field selection and a method invocation explicitly in the grammar. For instance, we could write the rule as follows:

```
// alternative DISCOURAGED implementation
SJSelectionExpression returns SJExpression:
  SJTerminalExpression
  (
    ({SJMethodInvocation.receiver=current} '.'
      method=[SJMethod]
        '(' (args+=SJExpression (',' args+=SJExpression)*)? ')')
    |
    ({SJFieldSelection.receiver=current} '.' field=[SJField])
  )* ;
```

After parsing a dot, the parser needs to scan all the way to the right parenthesis; if it finds it, then it creates an instance of `SJMethodInvocation` that refers directly to an `SJMethod`; otherwise, it parses the selection by creating an instance of `SJFieldSelection` which refers directly to an `SJField` object. This solution has the advantage that in the AST it is straightforward to distinguish a field selection from a method invocation since they are represented by objects of different types.

However, this version of the rule will require a parser that performs additional work, and most important of all, many IDE tooling features (such as the content assist) will not work as expected.

In general, a good practice in Xtext DSL implementations is to keep the grammar simple. It is better to have a loose grammar and a strict validation phase ("loose grammar, strict validation", see Zarnekow 2012).

Therefore, we keep the selection rule in its original form, and we will deal with the cases for field selection and method invocation in the validator and in other components.

Finally, we have the rule for terminal expressions:

```
SJTerminalExpression returns SJExpression:
  {SJStringConstant} value=STRING |
  {SJIntConstant} value=INT |
  {SJBoolConstant} value = ('true' | 'false') |
  {SJThis} 'this' |
  {SJNull} 'null' |
  {SJSymbolRef} symbol=[SJSymbol] |
  {SJNew} 'new' type=[SJClass] '(' ')' |
  '(' SJExpression ')'
;
```

Note that we have a single rule for referring both to a parameter (SJParamenter) and to a local variable (SJVariableDeclaration), since we have a cross-reference of type SJSymbol (see the rule we added previously to have a base class for both parameters and local variables). In our DSL you cannot pass arguments when creating an instance (SJNew) since we simplified the language by removing explicit parameterized constructors. Finally, as in the Expressions DSL, we have a rule that allows explicit parentheses.

As we hinted in the beginning of this chapter, member access always requires a receiver expression, even if it is this. If we also wanted to handle member access with an implicit receiver expression, we could have changed the rule for SJSymbol to also include SJMember, for example:

```
SJSymbol: SJVariableDeclaration | SJParameter | SJMember;
```

The features type and name are common to SJMember as well. However, this would require some additional work when implementing other aspects of the DSL, which might distract from the real intent of the SmallJava DSL, that is, being a case study for typing and scoping.

The complete grammar

We will sum up by showing the complete grammar of the SmallJava DSL (in the next chapter we will modify this grammar):

```
grammar org.example.smalljava.SmallJava with
  org.eclipse.xtext.common.Terminals

generate smallJava "http://www.example.org/smalljava/SmallJava"

SJProgram:
  classes+=SJClass*;
```

```
SJClass: 'class' name=ID ('extends' superclass=[SJClass])? '{'
  members += SJMember*
    '}' ;

SJMember:
  SJField | SJMethod ;

SJField:
  type=[SJClass] name=ID ';' ;

SJMethod:
  type=[SJClass] name=ID
    '(' (params+=SJParameter (',' params+=SJParameter)*)? ')'
    body=SJMethodBody ;

SJMethodBody:
  {SJMethodBody} '{' statements += SJStatement* '}';

SJStatement:
  SJVariableDeclaration |
  SJReturn |
  SJExpression ';' |
  SJIfStatement
;

SJVariableDeclaration:
  type=[SJClass] name=ID '=' expression=SJExpression ';'
;

SJReturn:
  'return' expression=SJExpression ';'
;

SJIfStatement:
  'if' '(' expression=SJExpression ')' thenBlock=SJIfBlock
  (=>'else' elseBlock=SJIfBlock)?
;

SJIfBlock:
  statements += SJStatement
  | '{' statements += SJStatement+ '}' ;

SJBlock: SJMethodBody | SJIfBlock ;

SJSymbol: SJVariableDeclaration | SJParameter ;

SJExpression: SJAssignment;
```

```
SJAssignment returns SJExpression:
  SJSelectionExpression
  ({SJAssignment.left=current} '=' right=SJExpression)?
;

SJSelectionExpression returns SJExpression:
  SJTerminalExpression
  (
    {SJMemberSelection.receiver=current} '.'
    member=[SJMember]
    (methodinvocation?='('
      (args+=SJExpression (',' args+=SJExpression)*)? ')'
    )?
  )* ;

SJTerminalExpression returns SJExpression:
  {SJStringConstant} value=STRING |
  {SJIntConstant} value=INT |
  {SJBoolConstant} value = ('true' | 'false') |
  {SJThis} 'this' |
  {SJNull} 'null' |
  {SJSymbolRef} symbol=[SJSymbol] |
  {SJNew} 'new' type=[SJClass] '(' ')' |
  '(' SJExpression ')'
;
```

Utility methods

As we did for previous DSLs, we write an Xtend class, `SmallJavaModelUtil`, with static utility methods for accessing the AST model of a SmallJava program:

```
import static extension org.eclipse.xtext.EcoreUtil2.*

class SmallJavaModelUtil {
  def static fields(SJClass c) {
    c.members.filter(typeof(SJField))
  }

  def static methods(SJClass c) {
    c.members.filter(typeof(SJMethod))
  }

  def static returnStatement(SJMethod m) {
    m.body.statements.typeSelect(typeof(SJReturn)).head
  }
```

```
  def static containingClass(EObject e) {
    e.getContainerOfType(typeof(SJClass))
  }

  def static containingBlock(EObject e) {
    e.getContainerOfType(typeof(SJBlock))
  }

  def static containingProgram(EObject e) {
    e.getContainerOfType(typeof(SJProgram))
  }

  def static containingMethod(EObject e) {
    e.getContainerOfType(typeof(SJMethod))
  }
}
```

Since the feature `members` in an `SJClass` object contains both the `SJField` and `SJMethod` instances, it is useful to have utility methods to quickly select them based on type. We will use these static methods in other Xtend classes with an **import static extension** statement so that we will be able to write expressions such as:

- `c.methods`
- `c.fields`

As we will see in the rest of the chapter, we will often need to directly access the containing class, the containing block, the containing program, and the containing method of an expression or statement thus, we write the corresponding utility methods (recall that `getContainerOfType` comes from `EcoreUtil2`, whose static methods are imported as extension methods). Finally, the method to quickly access the `return` statement will be useful when writing unit tests for the DSL.

Testing the grammar

As you should know by now, we should write unit tests for the parser as soon as we write some rules for the DSL grammar. In this chapter we show only a few interesting cases, in particular the tests for the associativity of expressions such as assignments and member selection (see *Chapter 8*, *An Expression Language*, for the technique for testing associativity). We use the `SmallJavaModelUtil` utility methods to write cleaner tests:

```
import static extension
  org.example.smalljava.util.SmallJavaModelUtil.*
import static extension org.junit.Assert.*
```

```
@RunWith(typeof(XtextRunner))
@InjectWith(typeof(SmallJavaInjectorProvider))
class SmallJavaParserTest {
  @Inject extension ParseHelper<SJProgram>
  @Inject extension ValidationTestHelper

  @Test def void testMemberSelectionLeftAssociativity() {
  '''
  class A {
    A m() { return this.m().m(); }
  }
  '''.parse.classes.head.methods.head.
        body.statements.last.assertAssociativity("((this.m).m)")
  }

  @Test def void testAssignmentRightAssociativity() {
  '''
  class A {
    A m() {
      A f = null;
      A g = null;
      f = g = null;
      return null;
    }
  }
  '''.parse.classes.head.methods.head.
     body.statements.get(2).assertAssociativity("(f = (g = null))")
}

  def private assertAssociativity(SJStatement s, CharSequence
expected) {
    expected.toString.assertEquals(s.stringRepr)
  }

  def private String stringRepr(SJStatement s) {
    switch (s) {
      SJAssignment:
        '''(«s.left.stringRepr» = «s.right.stringRepr»)'''
      SJMemberSelection:
        '''(«s.receiver.stringRepr».«s.member.name»)'''
      SJThis: "this"
      SJNew: '''new «s.type.name»()'''
      SJNull: "null"
```

```
        SJSymbolRef: s.symbol.name
        SJReturn: s.expression.stringRepr
    }
  }
```

The additional effort in writing tests for the SmallJava DSL is that when you need to access statements and expressions, you will need to walk the AST model to access the class, then the method, and then access the desired expression or statements in the method's body statement list. However, by relying on Xtend syntactic sugar, it is easy to write such logic.

First validation rules

Before getting to the main subject of this chapter, we will first implement some constraint checks that are complementary to type checking.

Checking cycles in class hierarchies

The SmallJava parser accepts input with cyclic class hierarchies, for instance:

```
class A extends C {}

class C extends B {}

class B extends A {}
```

This cannot be checked in the parser; we must write a validator `@Check` method to mark such situations as errors (this is similar to the validator for the Entities DSL in *Chapter 4*, *Validation*). As you will see later in this chapter, we will often need to traverse the inheritance hierarchy of a class to perform additional checks. Also in this case it is our job to avoid an infinite loop in case of cycles. We write a utility method in `SmallJavaModelUtil` that collects all the classes in the hierarchy of a given `SJClass`, to avoid entering an infinite loop. Similarly to what we did for the Entities DSL, we will visit the class hierarchy inspecting the `superclass` feature and we will stop the visit either when the superclass is `null` or when a class is already in the list of visited classes:

```
class SmallJavaModelUtil {

  def static classHierarchy(SJClass c) {
    val visited = <SJClass>newArrayList()

    var current = c.superclass
    while (current != null && !visited.contains(current)) {
      visited.add(current)
```

```
    current = current.superclass
  }

  visited
}...
```

> When looking at the example code, you will find that testing is done in isolation in its own Junit test class for SmallJavaModelUtil — this is not shown in this chapter.

Now it is straightforward to write the validator method for checking whether the hierarchy of a class contains a cycle; given an SJClass object, we just need to check whether that class is contained in its own class hierarchy.

```
import static extension
  org.example.smalljava.util.SmallJavaModelUtil.*

class SmallJavaValidator extends AbstractSmallJavaValidator {

  public static val HIERARCHY_CYCLE =
    "org.example.smalljava.HierarchyCycle";

  @Check
  def checkClassHierarchy(SJClass c) {
    if (c.classHierarchy.contains(c)) {
      error("cycle in hierarchy of class '" + c.name + "'",
        SmallJavaPackage::eINSTANCE.SJClass_Superclass,
        HIERARCHY_CYCLE, c.superclass.name)
    }
  }...
```

Differently from what we did in *Chapter 4, Validation* for the Entities DSL, we do not build the inheritance hierarchy directly in the validator; we implemented this functionality in SmallJavaModelUtil since we will use it in other parts.

We test this validator rule as follows:

```
@RunWith(typeof(XtextRunner))
@InjectWith(typeof(SmallJavaInjectorProvider))
class SmallJavaValidatorTest {
  @Inject extension ParseHelper<SJProgram>
  @Inject extension ValidationTestHelper

  @Test def void testClassHierarchyCycle() {
    '''
    class A extends C {}
```

```
    class C extends B {}
    class B extends A {}
    '''.parse => [
      assertHierarchyCycle("A")
      assertHierarchyCycle("B")
      assertHierarchyCycle("C")
    ]
}

def private void assertHierarchyCycle(SJProgram p,
  String expectedClassName) {
  p.assertError(
    SmallJavaPackage::eINSTANCE.SJClass,
    SmallJavaValidator::HIERARCHY_CYCLE,
    "cycle in hierarchy of class '" + expectedClassName + "'"
  )
}...
```

> Validation rules should be tested for both results; that is, a test for
> an input containing a validation error and a test for a valid input
> without validation errors. For example, you should also implement
> a test with an input without cycles in the hierarchy and use
> `assertNoErrors`. Due to a lack of space, we will skip these tests
> in this chapter.

Checking member selections

As we said in the section *Rules for Expressions*, we preferred to have a simple rule
for member selection; this means that we must check whether a member selection
actually refers to a field and whether a method invocation actually refers to a method
("Loose grammar, Strict validation"); in fact, this program, which is not valid,
is accepted by our DSL:

```
class C {
  C f;
  C m() {
    this.f(); // method invocation on a field
    this.m; // field selection on a method
    return null;
  }
}
```

We therefore write a `@Check` method for checking these situations. In particular, we use the boolean feature `methodinvocation` to know what we need to check; remember that this feature is set to true if the program text contains an opening parenthesis after the member reference. Without this feature, we would have had to inspect the actual text of the program.

```
public static val FIELD_SELECTION_ON_METHOD =
  "org.example.smalljava.FieldSelectionOnMethod"

public static val METHOD_INVOCATION_ON_FIELD =
  "org.example.smalljava.MethodInvocationOnField"

@Check
def void checkMemberSelection(SJMemberSelection sel) {
  val member = sel.member
  if (member != null) {
    if (member instanceof SJField && sel.methodinvocation)
      error(
        '''Method invocation on a field''',
        SmallJavaPackage::eINSTANCE.
          SJMemberSelection_Methodinvocation,
        METHOD_INVOCATION_ON_FIELD)
    if (member instanceof SJMethod && !sel.methodinvocation)
      error(
        '''Field selection on a method''',
        SmallJavaPackage::eINSTANCE.SJMemberSelection_Member,
        FIELD_SELECTION_ON_METHOD
      )
  }
}
```

Note that for an error concerning a method invocation on a field, we pass the `methodinvocation` feature when calling `error`. This way, the `error` marker will be placed on the opening parenthesis.

This validator rule is verified by the following two test methods:

```
@Test def void invocationOnField() {
  '''
  class A {
    A f;
    A m() {
      return this.f();
    }
  }
```

```
    '''.parse.assertError(
        SmallJavaPackage::eINSTANCE.SJMemberSelection,
        SmallJavaValidator::METHOD_INVOCATION_ON_FIELD,
        "Method invocation on a field"
    )
}

@Test def void fieldSelectionOnMethod() {
    '''
    class A {
        A m() {
            return this.m;
        }
    }
    '''.parse.assertError(
        SmallJavaPackage::eINSTANCE.SJMemberSelection,
        SmallJavaValidator::FIELD_SELECTION_ON_METHOD,
        "Field selection on a method"
    )
}
```

Checking return statements

We must check that a `return` statement is the last statement of a block. If there were other statements after the `return` statement, they would never be executed, and thus we should issue an "Unreachable code" error (like Java does). We could have enforced this in the grammar by writing the rule for `SJMethodBody` as follows (we only show the case for the block of the method body, since for the blocks of an `if` statement it is slightly more complicated):

```
SJMethodBody:
    {SJMethodBody} '{'
        statements += SJStatement*
        (returnStatement = SJReturn)?
    '}';

SJStatement:
    SJVariableDeclaration |
    SJExpression ';' |
    SJIfStatement
;
```

This solution has several drawbacks. First of all, while you can easily access the `return` statement of a block, you cannot treat all the statements of a block uniformly, and you will need to consider the case for the `return` statement separately. Most important of all, in case the `return` statement is not the last statement of a block, the error issued by the parser does not bring much information; indeed, several syntax errors will be generated that are hard to understand by the user (see the following screenshot). Not getting AST elements in such cases also means it is not possible (or very hard) to offer good quick fixes:
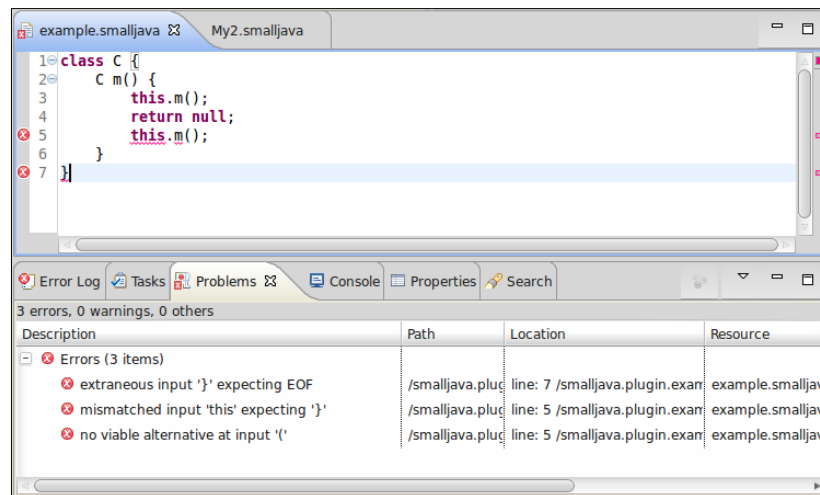
It is much better to have a loose grammar rule in combination with a strict check to provide better error messages. We will check that the `return` statement is the last element of a block's statement list; if this is not the case, we generate an error on the statement following the `return` statement, specifying that it is "Unreachable code" as shown in the following code snippet:

```
public static val UNREACHABLE_CODE =
  "org.example.smalljava.UnreachableCode"

@Check
def void checkNoStatementAfterReturn(SJReturn ret) {
  val statements = ret.containingBlock.statements
  if (statements.last != ret) {
    // put the error on the statement after the return
    error("Unreachable code",
      statements.get(statements.indexOf(ret)+1),
      null, // EStructuralFeature
      UNREACHABLE_CODE)
  }
}
```

We use the version of the `error` method that also accepts the `EObject` to mark with the error (if not specified, the object that is passed to the `@Check` method is marked). Since we pass `null` for the `EStructuralFeature` argument, the whole statement will be marked with the error.

To test that the error is actually generated on the correct statement, we rely on the type of the statement:

```
@Test def void testUnreachableCode() {
  '''
  class C {
    C m() {
      return null;
      this.m(); // the error should be placed here
    }
  }
  '''.parse.assertError(
    SmallJavaPackage::eINSTANCE.SJMemberSelection,
    SmallJavaValidator::UNREACHABLE_CODE,
    "Unreachable code"
  )
}
```

In the preceding code, we check that the error is on the statement of type `SJMemberSelection`, that is, the statement after the `return` statement in the test input code.

In the following screenshot, you can see how much better and informative the generated error will be:
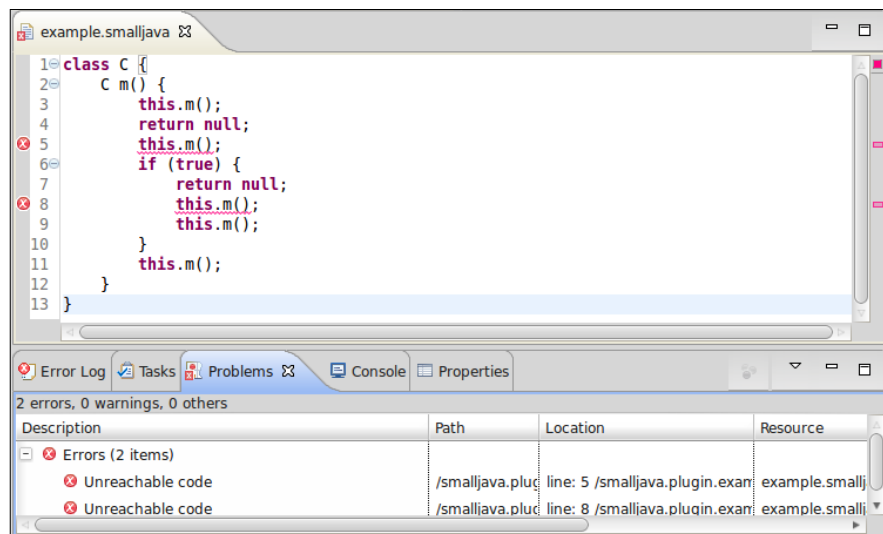
If we need a quick way of accessing the `return` statement of a block, we call the utility static method `returnStatement` in `SmallJavaModelUtil`.

The same utility method can be used to check that a method actually ends with a return statement (since in SmallJava there are no `void` methods); the implementation of this validator method can be found in the sources of the example. Also, in this case, the error contains more information than what we would get if we tried to rule out this situation in the grammar.

Checking for duplicates

In the DSL examples we have seen so far, we have always used the default validator `NamesAreUniqueValidator` to check for elements of the same kind with the same name. In this example, we will not enable this validator (by default it is not enabled in the MWE2 workflow) and we will deal with such checks manually in our own validator.

We want to allow methods and fields with the same name as in Java, and `NamesAreUniqueValidator` would mark this as an error. Moreover, `NamesAreUniqueValidator` relies on the Xtext index, which we are going to customize in an incompatible way at the end of the next chapter.

To check for duplicate elements, we search for an object that is different from the object being checked and that has the same name. We will use the same issue code for all errors related to duplicate elements.

For instance, for classes we write:

```
public static val DUPLICATE_ELEMENT =
  "org.example.smalljava.DuplicateElement"

@Check def void checkNoDuplicateClass(SJClass c) {
  if (c.containingProgram.classes.exists[
        it != c && it.name ==  c.name])
    error("Duplicate class '" + c.name + "'",
      SmallJavaPackage::eINSTANCE.SJClass_Name,
      DUPLICATE_ELEMENT)
}
```

In order to check that there are no duplicate fields and no duplicate methods, we write a single `@Check` method where we examine only the elements of the same kind as the `SJMember` being checked. This way, we will allow a field and a method to have the same name:

```
@Check def void checkNoDuplicateMember(SJMember member) {
  val duplicate = member.containingClass.members.findFirst[
      it != member && it.eClass == member.eClass &&
      it.name == member.name]
  if (duplicate != null)
    error("Duplicate member '" + member.name + "'",
      SmallJavaPackage::eINSTANCE.SJMember_Name,
      DUPLICATE_ELEMENT)
}
```

We can safely check for equality on the `EClass` of the members since EMF guarantees that for each `EClass` there is only one instance in the system.

For members, we write tests to verify error situations and valid programs:

```
@Test def void testDuplicateMethods() {
  '''
  class C {
    C m() { return null; }
    C m() { return null; }
  }
  '''.parse.assertError(
      SmallJavaPackage::eINSTANCE.SJMethod,
      SmallJavaValidator::DUPLICATE_ELEMENT,
      "Duplicate member 'm'"
    )
}

@Test def void testDuplicateFields() {
  '''
  class C {
    C f;
    C f;
  }
  '''.parse.assertError(
      SmallJavaPackage::eINSTANCE.SJField,
      SmallJavaValidator::DUPLICATE_ELEMENT,
      "Duplicate member 'f'"
    )
```

```
    }

    @Test def void testFieldAndMethodWithTheSameNameAreOK() {
      '''
      class C {
        C f;
        C f() { return null; }
      }
      '''.parse.assertNoErrors
    }
```

The check for duplicate variable declarations must deal with possible nested blocks, since we do not allow a local variable to shadow a previously defined variable with the same name. Instead of inspecting all possible nested blocks manually, we use the method `EcoreUtil2.getAllContentsOfType`, which returns the list of all elements of a given type, even the nested ones:

```
@Check
def void checkNoDuplicateVariable(SJVariableDeclaration vardecl) {
  val duplicate = vardecl.containingMethod.body.
    getAllContentsOfType(typeof(SJVariableDeclaration)).
      findFirst[it != vardecl && it.name == vardecl.name]
  if (duplicate != null)
    error("Duplicate variable declaration '" + vardecl.name + "'",
      SmallJavaPackage::eINSTANCE.SJSymbol_Name,
      DUPLICATE_ELEMENT)
}
```

Type checking

Most of the constraint checks for an object-oriented language such as SmallJava will deal with **type checking**; that is, checking that expressions and statements are well-typed.

We have already seen how to perform a simple form of type checking in the Expressions DSL (*Chapter 8*, *An Expression Language*). In this chapter, we will see an advanced type checking mechanism which includes **Subtyping** or **type conformance**: an object of class C can be used in a context where an object of a superclass of C is expected.

We will follow the same strategy illustrated in *Chapter 8*, *An Expression Language*: we will separate the type computation from the actual type checking. We will be able to generate the error on the sub-expression or statement that is the source of the problem.

Type provider for SmallJava

The type provider for SmallJava expressions we are about to construct will compute the type of any `SJExpression`. The concept of type will be represented by `SJClass` since SmallJava does not support native types (such as `int`, `boolean`, and so on).

We write a single `typeFor` method which returns an `SJClass` object by using a type switch (the default case simply returns `null`).

```
import static extension org.eclipse.xtext.EcoreUtil2.*

class SmallJavaTypeProvider {
  def typeFor(SJExpression e) {
    switch (e) {
      SJNew: e.type
      SJSymbolRef: e.symbol?.type
      SJMemberSelection: e.member?.type
      SJThis : e.getContainerOfType(typeof(SJClass))
      ...
    }
  }...
```

In the preceding method, the type of a new instance expression is clearly the class that we are instantiating (the feature: `type`). The type of a symbol reference is the type of the referred symbol. Similarly, the type of a member reference is the type of the referred member. The type for `this` is simply the type of the containing class. Note that while at runtime the actual object replacing `this` could be an object of a subclass, statically, its type is always the class where `this` is being used. In all of the preceding cases, the type always corresponds to an existing `SJClass`.

Now we need to provide a type for the remaining terminal expressions, that is, `null` and the constant expressions. For these expressions, there are no existing `SJClass` instances that we can use as types: we will create static instances in `SmallJavaTypeProvider` (for convenience, we will also give them a name):

```
class SmallJavaTypeProvider {
  public static val stringType =
  SmallJavaFactory::eINSTANCE.createSJClass => [name='stringType']
  public static val intType =
  SmallJavaFactory::eINSTANCE.createSJClass => [name = 'intType']
  public static val booleanType =
  SmallJavaFactory::eINSTANCE.createSJClass =>[name='booleanType']

  public static val nullType =
  SmallJavaFactory::eINSTANCE.createSJClass => [name = 'nullType']
```

```
    def typeFor(SJExpression e) {
      switch (e) {
        ...continuation
        SJNull: nullType
        SJStringConstant: stringType
        SJIntConstant: intType
        SJBoolConstant: booleanType
      }
    }

    def isPrimitive(SJClass c) {
      c.eResource == null
    }
  ...
```

Note that it is convenient to have a way of identifying the types we created for `null` and for constant expressions, which we call **primitive types**; we have a specific method for that called `isPrimitive`. An easy way to identify such types is to check that they are not part of a resource. We will need this distinction in the next chapter.

To test the type provider and keep the tests clean and compact, we implement a method that contains the skeleton of the test logic where a single passed expression is replaced. This way, the actual test methods are compact and simple since they only specify the expression and the expected type name:

```
@RunWith(typeof(XtextRunner))
@InjectWith(typeof(SmallJavaInjectorProvider))
class SmallJavaTypeProviderTest {
  @Inject extension ParseHelper<SJProgram>
  @Inject extension ValidationTestHelper
  @Inject extension SmallJavaTypeProvider

  def private assertType(CharSequence testExp,
                         String expectedClassName) {
  '''
  class R { public V v; }
  class P extends R { public R m() { return null; } }
  class V extends R { public N n; }
  class N extends R {}
  class F extends R {}

  class C extends R {
    F f;

    R m(P p) {
```

```
        V v = null;
        «testExp»;
        return null;
    }
  }
  '''.parse => [
      assertNoErrors
      expectedClassName.assertEquals(
        classes.last.methods.last.
          body.statements.get(1).statementExpressionType.name
      )
    ]
  }

  def private statementExpressionType(SJStatement s) {
    (s as SJExpression).typeFor
  }

  @Test def void thisType() {"this".assertType("C")}
  @Test def void paramRefType() {"p".assertType("P")}
  @Test def void varRefType() {"v".assertType("V")   }
  @Test def void newType() {"new N()".assertType("N")}
  @Test def void fieldSelectionType() {"this.f".assertType("F")}
...other cases...
  @Test def void intConstantType() {'10'.assertType("intType")}
  @Test def void nullType() {'null'.assertType("nullType")}
```

This technique is useful when you need a complete program to perform tests. In SmallJava you cannot type an expression without having a containing method and a containing class.

Type conformance (subtyping)

In an object-oriented language, the type system must also take **type conformance** (or **Subtyping**) into account: an object of class C can be used in a context where an object of a superclass of C is expected. For instance, the following code is well-typed:

```
C c = new D();
```

provided that D is a subclass (subtype) of C. This holds true in every context where an expression is assigned, for example, when we pass an argument in a method invocation.

We implement type conformance in a separate class, `SmallJavaTypeConformance`.

To check whether a class is a subclass of another class, we need to inspect the class hierarchy of the former and see whether we find the latter. In `SmallJavaModelUtil`, we already have a method that computes the class hierarchy avoiding infinite loops in case of a cyclic hierarchy, and we implement the method `isSubclassOf` as follows:

```
import static extension org.example.smalljava.util.
SmallJavaModelUtil.*

class SmallJavaTypeConformance {

  def isSubclassOf(SJClass c1, SJClass c2) {
    c1.classHierarchy.contains(c2)
  }...
```

Type conformance deals with subclasses as well as other special cases. For instance, a class is not considered a subclass of itself, but it is of course conformant to itself. Another special case is the expression `null`; it can be assigned to any variable and field and passed as an argument for any parameter. The type for `null`, which is represented by the static instance `nullType` in `SmallJavaTypeProvider`, must be conformant to any other type.

This is the initial implementation of type conformance for SmallJava:

```
import static org.example.smalljava.typing.SmallJavaTypeProvider.*

class SmallJavaTypeConformance {
  ...
  def isConformant(SJClass c1, SJClass c2) {
    c1 == nullType || // null can be assigned to everything
    c1 == c2 ||
    c1.isSubclassOf(c2)
  }...
```

For the moment, we are not considering other cases (we need some additional concepts, as we will see in the next chapter).

We test this implementation as follows:

```
@RunWith(typeof(XtextRunner))
@InjectWith(typeof(SmallJavaInjectorProvider))
class SmallJavaTypeConformanceTest {
  @Inject extension ParseHelper<SJProgram>
  @Inject extension SmallJavaTypeConformance
  @Inject extension SmallJavaTypeProvider
```

```
@Test def void testClassConformance() {
'''
class A {}
class B extends A {}
class C {}
'''.parse.classes => [
    // A subclass of A
    get(0).isConformant(get(0)).assertTrue
    // B subclass of A
    get(1).isConformant(get(0)).assertTrue
    // C not subclass of A
    get(2).isConformant(get(0)).assertFalse
  ]
}

@Test def void testNullConformance() {
'''
class C {}
class D {
  C m() { return null; }
}
'''.parse.classes => [
    val typeOfNull = last.methods.head.
          returnStatement.expression.typeFor
    // null can be assigned to anything
    typeOfNull.isConformant(get(0)).assertTrue
    typeOfNull.isConformant(get(1)).assertTrue
  ]
}...
```

Expected types

Now we must check whether an expression has a type which conforms to the one expected by the context where it is used. The context is not necessarily an assignment or a method invocation; for instance, the expression used in an `if` statement must have the type boolean.

An obvious but not very good way of implementing this check is to write a `@Check` method in the validator for each specific context where the conformance needs to be checked. For example, the check for the assignment expression could be implemented as shown in the following code snippet:

```
@Check
def void checkAssignment(SJAssignment a) {
  val actualType = a.right.typeFor
  val expectedType = a.left.typeFor
```

```
    if (!actualType.isConformant(expectedType)) {
      error(...
    }
  }
```

However, if we followed this approach, we would need to write several methods in the validator, which all have the preceding logic in common.

It is easier to compute the **expected type** and **actual type** of an expression separately and then check for conformance rather than checking explicitly for each language construct. The expected type of an expression depends on the context where the expression is being used. We implement the method, expectedType, in SmallJavaTypeProvider.

The idea is that, given an expression, its expected type depends on its role in the container of the expression. For instance, consider this SJVariableDeclaration:

```
    C c = new D();
```

The expected type of the expression new D() depends on the fact that it is contained in a variable declaration; the role, in particular, is represented by the feature of the container that contains the expression; in this example, the feature of SJVariableDeclaration is expression. We get the container using the method eContainer and the containing feature using the method eContainingFeature. Then, it is just a matter of dealing with all the cases. We will use a typed switch that allows us to specify additional conditions (using the keyword case). Note that the typed switch is performed not on the expression, but on the container of the expression, and the case part deals with the containing feature. This reflects the fact that the expected type of an expression does not depend on the expression itself, but on the containing context.

To summarize, when the expression:

- is the initialization expression of a variable declaration, the expected type is the type of the declared variable
- is the right-hand side of an assignment, the expected type is the type of the left-hand side of the assignment
- is the expression of a return statement, the expected type is the return type of the containing method
- is the expression of an if statement, the expected type is boolean
- is an argument of a method invocation, the expected type is the type of the corresponding parameter of the invoked method.

The implementation is as follows:

```
val ep = SmallJavaPackage::eINSTANCE

def expectedType(SJExpression e) {
  val c = e.eContainer
  val f = e.eContainingFeature
  switch (c) {
    SJVariableDeclaration case f ==
      ep.SJVariableDeclaration_Expression : c.type
    SJAssignment case f == ep.SJAssignment_Right :
      c.left.typeFor
    SJReturn case f == ep.SJReturn_Expression :
      c.containingMethod.type
    case f == ep.SJIfStatement_Expression:
      booleanType
    SJMemberSelection case f == ep.SJMemberSelection_Args : {
      // assume that it refers to a method and that there
      // is a parameter corresponding to the argument
      try {
        (c.member as SJMethod).params.get(c.args.indexOf(e)).type
      } catch (Throwable t) {
        null // otherwise there is no specific expected type
      }
    }
  }
}
```

Note that in the last case, we enclose the computation of the expected type in a try-catch; in fact, there are some things that can go wrong in this case: the invoked member does not exist, it is not a method, or there is no parameter corresponding to the argument. In these cases, an exception is thrown and we simply return `null` as the expected type. If this happens, the corresponding error situation is reported by other validation checks.

Checking type conformance

Now we are able to write a single validator method to check type conformance for a generic expression:

```
class SmallJavaValidator extends AbstractSmallJavaValidator {
  @Inject extension SmallJavaTypeProvider
  @Inject extension SmallJavaTypeConformance
```

```
    public static val INCOMPATIBLE_TYPES =
      "org.example.smalljava.IncompatibleTypes"

    @Check
    def void checkCompatibleTypes(SJExpression exp) {
      val actualType = exp.typeFor
      val expectedType = exp.expectedType
      if (expectedType == null || actualType == null)
        return; // nothing to check
      if (!actualType.isConformant(expectedType)) {
        error("Incompatible types. Expected '" + expectedType?.name
          + "' but was '" + actualType?.name + "'", null,
          INCOMPATIBLE_TYPES);
      }
    }
```

To test this method in `SmallJavaValidatorTest`, we must create a test for each situation where conformance is not respected; as we did for the type provider, we write a method that contains the skeleton of the input, where the passed expression or statement is replaced:

```
    def void assertIncompatibleTypes(CharSequence methodBody,
                EClass c, String expectedType, String actualType) {
      '''
      class F {}
      class R {}
      class P {}
      class P1 {}
      class P2 {}
      class V {}
      class C {
        F f;
        R m(P p) {
          «methodBody»
        }
        R n(P1 p1, P2 p2) { return null; }
      }
      '''.parse.assertError(
          c, SmallJavaValidator::INCOMPATIBLE_TYPES,
          "Incompatible types. Expected '" + expectedType
            + "' but was '" + actualType + "'"
      )
    }
```

```
@Test def void assertVariableDeclExpIncompatibleTypes() {
  "V v = new P();".
    assertIncompatibleTypes(
      SmallJavaPackage::eINSTANCE.SJNew,"V", "P")
}

@Test def void assertReturnExpIncompatibleTypes() {
  "return p;".
    assertIncompatibleTypes(
      SmallJavaPackage::eINSTANCE.SJSymbolRef, "R", "P")
}

@Test def void assertArgExpIncompatibleTypes() {
  "this.n(new F(), new V());" => [
    assertIncompatibleTypes(
      SmallJavaPackage::eINSTANCE.SJNew, "P1", "F")
    assertIncompatibleTypes(
      SmallJavaPackage::eINSTANCE.SJNew, "P2", "V")
  ]
}...other cases not shown...
```

We still need to check whether the number of arguments passed to a method invocation is equal to the number of parameters of the invoked method:

```
@Check
def void checkMethodInvocationArguments(SJMemberSelection sel) {
  if (sel.member != null && sel.member instanceof SJMethod) {
    val method = sel.member as SJMethod
    if (method.params.size != sel.args.size) {
      error(
        "Invalid number of arguments. The method " +
        method.memberAsStringWithType +
        " is not applicable for the arguments " +
        sel.argsTypesAsStrings,
        SmallJavaPackage::eINSTANCE.SJMemberSelection_Member,
        INVALID_ARGS)
    }
  }
}
```

We implemented some utility methods (not shown here) to give a representation of a method signature together with its arguments and return type (`memberAsStringWithType`) and of the list of argument types (`argsTypesAsStrings`), so that we can produce a better error message.

The following screenshot shows some validation errors:
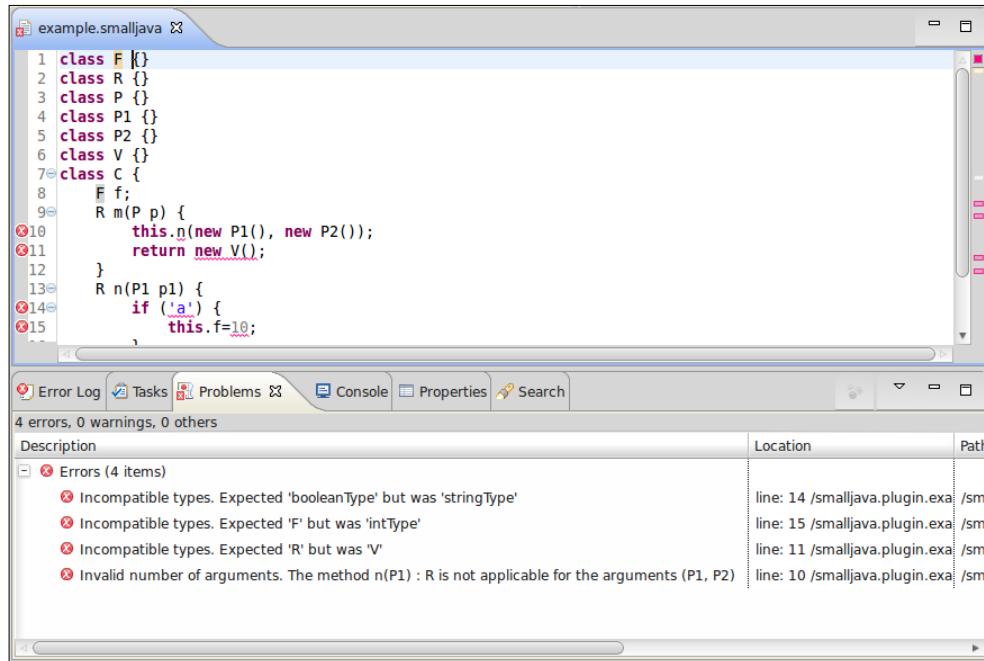
Checking method overriding

Finally, we must check that method overrides are correct: the return type must be conformant to the type of the overridden method and the parameter types must be the same as the ones of the overridden method (of course, the parameter names can be different):

```
@Check
def void checkMethodOverride(SJMethod m) {
  val overridden = m.containingClass.classHierarchy.
    map[methods].flatten.findFirst[it.name==m.name]

  if (overridden != null) {
    if (!m.type.isConformant(overridden.type) ||
      !m.params.map[type].
        elementsEqual(overridden.params.map[type])) {
      error("The method must override a superclass method",
        SmallJavaPackage::eINSTANCE.SJMember_Type,
        WRONG_METHOD_OVERRIDE)
    }
  }
}
```

Let us do a partial examination of this implementation. First of all, we get the list of the lists of methods in the class hierarchy of the containing class; we combine this list of lists of methods into a single list (actually an `Iterable` instance) of methods using the standard library utility method `flatten`. Then, we search for a method with the same name as the examined method. If we find it, we check that the return type of the examined method is conformant with the one of the overridden method and that the parameter types are the same. For this latter check, we get the list of types of parameters of the two methods and we use the standard library utility method `elementsEqual`; this checks whether the two Iterables have the same number of elements and that each element of the first Iterable is equal to the corresponding element of the other Iterable.

> This implementation is potentially slow when dealing with large models (due to its quadratic complexity). You might want to later optimize it by aggregating methods in a map with appropriate keys to find associated methods much faster. Once you have a bunch of tests for the simple implementation, you can experiment with optimizations and make sure that tests still succeed.

Improving the UI

To provide a better experience to the user of the SmallJava editor and tooling, we customize the appearance of SmallJava members (fields and methods) in several places of the UI.

First of all, we give a better string representation of members by also showing their `type` feature; thus, the string representation of a member is its name and its type's name separated by a colon `:`. Moreover, for methods, we also show the type of each parameter. We mimic the representation of Java members as implemented by Eclipse JDT. We then implement the methods for string representation in `SmallJavaModelUtil`:

```
def static memberAsString(SJMember m) {
  switch (m) {
    SJField: m.name
    SJMethod: m.name + "(" +
      m.params.map[type?.name].join(", ") + ")"
  }
}

def static memberAsStringWithType(SJMember m) {
  m.memberAsString + " : " + m.type.name
}
```

We also borrow icons from Eclipse JDT for classes, fields, and methods. As we saw in *Chapter 6, Customizations,* we can specify the label representation for our DSL model elements by implementing text and image methods in SmallJavaLabelProvider. When we implement text, we can return a JFace StyledString object, which allows us to also specify the style (font, color, and so on) of the resulting label. For instance, we want to represent the part starting with : using a different style, again to mimic JDT:

```
class SmallJavaLabelProvider extends
  org.eclipse.xtext.ui.label.DefaultEObjectLabelProvider {

  def text(SJMethod m) {
    new StyledString(m.memberAsString).
      append(new StyledString(" : " + m.type.name,
        StyledString::DECORATIONS_STYLER))
  }

  def text(SJField f) {
    new StyledString(f.name).
      append(new StyledString(" : " + f.type.name,
        StyledString::DECORATIONS_STYLER))
  }...
```

We also want a custom representation of members when they are proposed by the content assist; this will provide the user with additional information about the available members. In our SmallJavaProposalProvider class, we can simply override the getStyledDisplayString method that is automatically called by the default implementation of the proposal provider to represent the proposals. In this case, we return a custom StyledString object for representing an SJMember element, and we use a different style for representing the class containing the proposed member:

```
class SmallJavaProposalProvider extends
          AbstractSmallJavaProposalProvider {

  override getStyledDisplayString(EObject element,
          String qualifiedNameAsString, String shortName) {
    if (element instanceof SJMember) {
      val member = element as SJMember
```

```
        new StyledString(member.memberAsStringWithType).
        append(new StyledString(" - " + member.containingClass.name,
            StyledString::QUALIFIER_STYLER))
      } else
      super.getStyledDisplayString(element,
            qualifiedNameAsString, shortName)
    }...
```

The result can be seen in the following screenshot (we also customized the outline view, as we did in *Chapter 6, Customizations*). Note all the type related information that is now available in the UI:
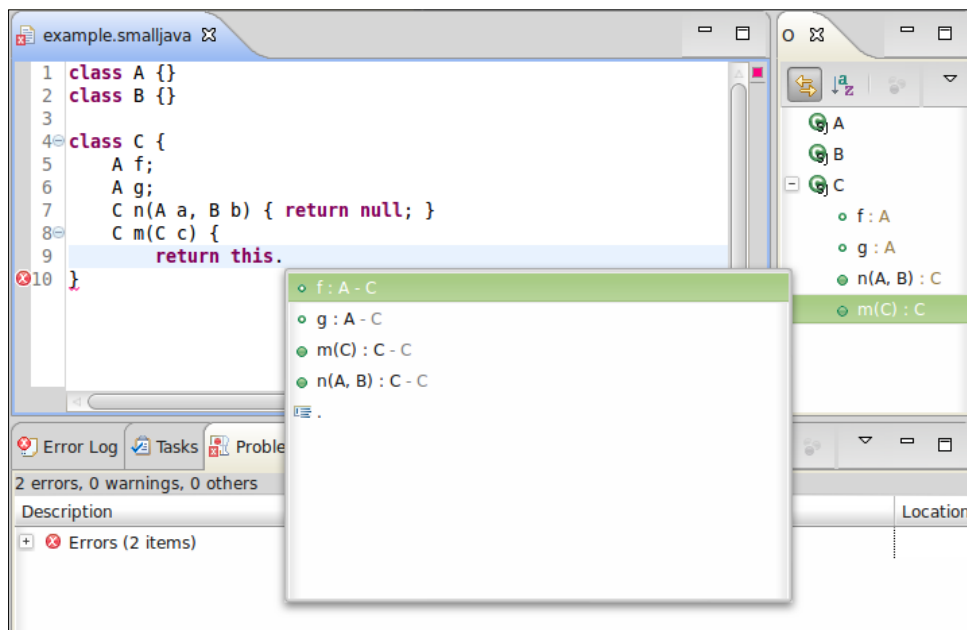
Summary

In this chapter we presented type checking techniques that are typical for a DSL with object-oriented features. A small Java-like language was introduced to demonstrate how to parse features such as member access and inheritance and how to handle validation of type conformance.

There is however a crucial aspect that we still have to deal with: correct access to members (fields and methods). In fact, the selection expression

```
e.f
```

is well-typed only if the field `f` is declared in the class of `e` (similarly for methods) or in any superclass of the class of `e`. If you perform some experiments, you will note that at the moment, you can access members which are not declared in the class of the receiver expression, and that you cannot access all the members of the hierarchy of the class of the receiver expression. Furthermore, local variable access does not work correctly in the current implementation: you can also refer to variables defined later and variables defined in inner blocks.

In order to correctly deal with the preceding issues, which concern cross-reference resolution, we will need to implement a custom scoping mechanism, as we will see in the next chapter. Scoping defines what is visible in a specific context so that Xtext can correctly resolve cross-references.

In the next chapter, we will also add to SmallJava access level modifiers for members (that is, `private`, `protected`, and `public`). We will show you how a SmallJava program can access classes defined in other files and how to provide a library with some predefined classes (for example, `Object`, `String`, and so on).

10
Scoping

As we have seen in the previous chapters, the first aspect you need to customize in your DSL implementation in Xtext is the validator. Typically, the second aspect you need to customize is **scoping**, which is the main mechanism behind visibility and cross-reference resolution. As soon as the DSL needs some constructs for structuring the code, a custom scoping implementation is required. In particular, scoping and typing are often strictly connected and interdependent especially for object-oriented languages. For this reason, in this chapter, we will continue developing the SmallJava DSL introduced in the previous chapter. We will describe both local and global scoping and explain how to customize scoping using SmallJava as a case study.

All the components that deal with scoping are fully documented in the Xtext documentation. However, it looks like scoping is one of the hardest topics to fully understand when developing with Xtext (it is one of the most common subjects in the Xtext forum). This is due to the fact that some knowledge about the infrastructure of Xtext and how scoping is used is required to fully understand it. In this chapter, we will try to give a presentation of these mechanisms in a way that should help you implement a custom scoping correctly.

Cross-reference resolution in Xtext

Cross-reference resolution involves several mechanisms. In this section we introduce the main concepts behind these mechanisms and describe how they interact. We will also write tests to get familiar with cross-reference resolution.

Containments and cross-references

Xtext relies on EMF for the in-memory representation of a parsed program, thus, it is necessary to understand how cross-references are represented in EMF.

In EMF, when a feature is not of the basic type (string, integer, and so on), it is actually a reference (an instance of `EReference`). A **containment reference** defines a stronger type of association. The referenced object is contained in the referring object, called the **container**; in particular, an object can have only one container. For non-containment references, the referenced object is stored somewhere else, for example, in another object of the same resource or even in a different resource. A cross-reference is implemented as a non-containment reference.

> In Ecore the `EReference` class has a property called `containment` that defines whether the reference is a containment reference or not.

Let us recall the SmallJava DSL rule for a class definition:

```
SJClass: 'class' name=ID ('extends' superclass=[SJClass])? '{'
    members += SJMember*
        '}' ;
```

An `SJMember` member is contained in an `SJClass` class, that is, `members` is a multi-value containment reference; on the contrary, `superclass` is a cross-reference.

Scoping deals with cross-references, that is, references with containment set to false.

The index

By default, in a DSL implemented in Xtext, every object which can be given a name is automatically visible in a program, which means it can be referred to.

This mechanism is handled in Xtext by using an **index**. The index stores information about all the objects in every resource. For technical reasons (mainly efficiency and memory overhead) the index stores the `IEObjectDescription` elements instead of the actual objects. An `IEObjectDescription` element is an abstract description of `EObject`. This description contains the name of the object and the EMF URI of the object. The EMF URI is a path that includes the resource of the object and its position within the resource; the URI allows to locate and load the actual object when needed. The description also contains the `EClass` of the object, which is useful to filter descriptions by type.

The set of resources handled by the index depends on the context of execution. In the IDE, Xtext indexes all the resources in all the projects with the Xtext nature; the index is kept up-to-date in an incremental way using the incremental building mechanism of Eclipse, thus keeping the overhead minimal.

In a plain runtime context, where there is no workspace, the index is based on the EMF `ResourceSet`. We will see what this implies when we write Junit tests and when we implement a standalone compiler.

In both contexts the index is global; visibility across resources is handled by using containers as shown later in the section, *The index and the containers*.

Qualified names

The default implementation of the index uses a mechanism based on names. The computation of names is delegated to `IQualifiedNameProvider`. The default implementation of the name provider is based on the feature `name`, this is why we always defined elements that we want to refer to with a feature `name` in the grammar.

Using only the plain name will soon raise problems due to duplicates, even in a simple program. Most Java-like languages use **namespaces** to allow for elements with the same name, provided they are in different namespaces. Thus, for example, two different methods are allowed to have local variables with the same name, two different classes are allowed to have fields with the same name, two different packages are allowed to have classes with the same name, and so on. For this reason, the default implementation of the name provider computes a **qualified name**. It concatenates the name of an element with the qualified name of its container recursively; elements without a name are simply skipped. By default, all segments of a qualified name are separated by a dot, which is a common notation for expressing qualified names like, for example, in Java.

For example, consider the following SmallJava program (the type of declarations is not important here, thus, we use the class A that we assume as defined in the program):

```
class C {
  A f;
  A m(A p) {
    A v = null;
    return null;
  }
}
```

The containment relations are shown in the following tree figure:
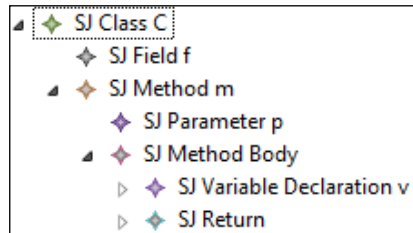
The qualified names of the elements of the preceding SmallJava class are shown in the following table:

Object	Qualified Name
SJClass C	C
SJField f	C.f
SJMethod m	C.m
SJParameter p	C.m.p
SJVariableDeclaration v	C.m.v

Note that `SJMethodBody` does not participate in the computation of the qualified name of the contained variable declaration, since it does not have a name feature.

> Qualified names are the mechanism also used by `NamesAreUniqueValidator` to decide when two elements are considered as duplicates

Exported objects

To understand the mechanism behind scoping, it is useful to learn how to access the index. Accessing the index will also be useful for performing additional checks as shown later in this chapter. We show how to get all the object descriptions of the current program, that is, of the current resource.

The indexed object descriptions of a resource are stored in `IResourceDescription`, which is an abstract description of a resource.

The index is implemented by `IResourceDescriptions` (plural form) and can be obtained through an injected `ResourceDescriptionsProvider` using the method `getResourceDescriptions(Resource)`.

Once we have the index, we can get `IResourceDescription` of a resource by specifying its URI.

Once we have `IResourceDescription`, we get the list of all the `IEObjectDescription` elements of the resource that are externally visible, that is, globally exported, using the method `getExportedObjects`.

We implement all the index related operations in a separate class, `SmallJavaIndex`:

```
class SmallJavaIndex {
  @Inject ResourceDescriptionsProvider rdp
  def  getResourceDescription(EObject o) {
    val index = rdp.getResourceDescriptions(o.eResource)
    index.getResourceDescription(o.eResource.URI)
  }

  def getExportedEObjectDescriptions(EObject o) {
    o.getResourceDescription.getExportedObjects
  }
}
```

We can then write a test (see *Chapter 7, Testing*) listing the qualified names of the exported object descriptions:

```
@RunWith(typeof(XtextRunner))
@InjectWith(typeof(SmallJavaInjectorProvider))
class SmallJavaIndexTest {
  @Inject extension ParseHelper<SJProgram>
  @Inject extension SmallJavaIndex
  @Test def void testExportedEObjectDescriptions() {
    '''
    class C {
      A f;
      A m(A p) {
        A v = null;
        return null;
      }
    }
    class A {}
    '''.parse.assertExportedEObjectDescriptions
        ("C, C.f, C.m, C.m.p, C.m.v, A")
  }
  def private assertExportedEObjectDescriptions(EObject o,
                    CharSequence expected) {
   expected.toString.assertEquals(
    o.getExportedEObjectDescriptions.map[qualifiedName].join(", ")
   )
  }
}
```

The linker and the scope provider

The actual cross-reference resolution, that is, the linking, is performed by `LinkingService`. Usually, you do not need to customize the linking service, since the default implementation relies on `IScopeProvider`, which is the component you often need to customize. The default linking service relies on the scope obtained for a specific context in the model, and chooses an object whose name matches the textual representation of the reference in the program.

Thus, a scope provides information about:

- The objects that can be reached, that is, they are visible, in a specific part of your model (the context)
- The textual representation to refer to them

You can think of a scope as a symbol table (or a map) where the keys are strings and the values are instances of `IEObjectDescription`. The Java interface for scopes is `IScope`.

The overall process of cross-reference resolution, that is, the interaction between the linker and the scope provider, can be simplified as follows:

1. The linker must resolve a cross-reference with text `n` in the program context `c` for the feature `f` of type `t`.
2. The linker queries the scope provider: "give me the scope for the elements assignable to `f` in the program context `c`".
3. The linker searches in the obtained scope for an element whose key matches with `n`.
4. If it finds it, it resolves the cross-reference and performs the linking.
5. Otherwise, it issues an error of the type "Couldn't resolve reference to...".

The `IScopeProvider` entry point is a single method:

```
IScope getScope(EObject context, EReference reference)
```

Since the program is stored in an EMF model, the context is `EObject` and the reference is `EReference`. Note that cross-references in an Xtext grammar specify the type of the referred elements. Thus, the scope provider must also take the types of objects for that specific reference into consideration when building the scope. The type information is retrieved from `EReference` passed to `getScope`.

Both scopes and the index deal with object descriptions. The crucial difference is that a scope also specifies the actual string with which you can refer to an object. The actual string does not have to be equal to the object description's qualified name in the index. Thus, the same object can be referred to with different strings in different program contexts.

To put it in another way, the index provides all the qualified names of the visible objects of a resource so that all these objects can be referred to using their qualified names. The scope provides further information , that is, in a given program context, some objects can be referred to even using simple names or using qualified names with less segments.

If in our DSL we can only use IDs to refer to objects, and objects are visible only by their fully qualified names, then we will not be able to refer to any object. Thus, being able to refer to an object by a simple name is essential.

Another important aspect of scopes is that they can be nested. Usually, a scope is part of a list of scopes, so that a scope can have an outer scope, also known as parent scope. If a matching string cannot be found in a scope, it is recursively searched in the outer scope. If a matching string is found in a scope, the outer scope is not consulted. This strategy allows Xtext to implement typical situations in programming languages. For example, in a code block, you can refer to variables declared in the containing block. Similarly, declarations in a block usually shadow possible declarations with the same name in the outer context.

The default implementation of `IScopeProvider` reflects the nested nature of scopes by using the containment relations of the EMF model. For a given program context, it builds a scope where all objects in the container of the context are visible by their simple name; the outer scope is obtained by recursively applying the same strategy on the container of the context.

Let us go back to our previous SmallJava example:

```
class C {
  A f;
  A m(A p) {
    A v = null;
    return null; // assume this is the context
  }
}
```

Let us assume that the context is the expression of the return statement; in that context the SJMember elements will be visible by simple names and by qualified names:

```
f, m, C.f, C.m
```

In the same context, the SJSymbol elements will be visible by the following names:

```
p, v, m.p, m.v, C.m.p, C.m.v
```

This is because p and v are contained in m which in turn is contained in C. Note that they are also visible by simpler qualified names, that is, qualified names with less fragments, m.p and m.v, respectively.

In the SmallJava grammar, we can refer to members and symbols only by their simple name using an ID, not by a qualified name. Without a scope we would not be able to refer to any of such elements, since they would be visible only by their qualified names.

Again, a learning test that invokes the method getScope can help understand the default scope provider implementation (note how we specify EReference by using SmallJavaPackage):

```
@RunWith(typeof(XtextRunner))
@InjectWith(typeof(SmallJavaInjectorProvider))
class SmallJavaScopeProviderTest {
  @Inject extension ParseHelper<SJProgram>
  @Inject extension ValidationTestHelper
  @Inject extension IScopeProvider
  @Test def void testScopeProvider() {
    '''
    class C {
      A f;
      A m(A p) {
        A v = null;
        return null;
      }
    }
    class A {}
    '''.parse.classes.head.
        methods.last.returnStatement.expression => [
          assertScope
          (SmallJavaPackage::eINSTANCE.SJMemberSelection_Member,
            "f, m, C.f, C.m")
          assertScope
```

```
            (SmallJavaPackage::eINSTANCE.SJSymbolRef_Symbol,
               "p, v, m.p, m.v, C.m.p, C.m.v")
          ]
      }

    def private assertScope(EObject context,
                   EReference reference, CharSequence expected) {
      context.assertNoErrors
      expected.toString.assertEquals(
        context.getScope(reference).
          allElements.map[name].join(", "))
    }
  }
```

The program context of the preceding test is the expression of the return statement.

In the next diagram, we show the interaction between the linker and the scope provider for the resolution of the member in a member selection expression:
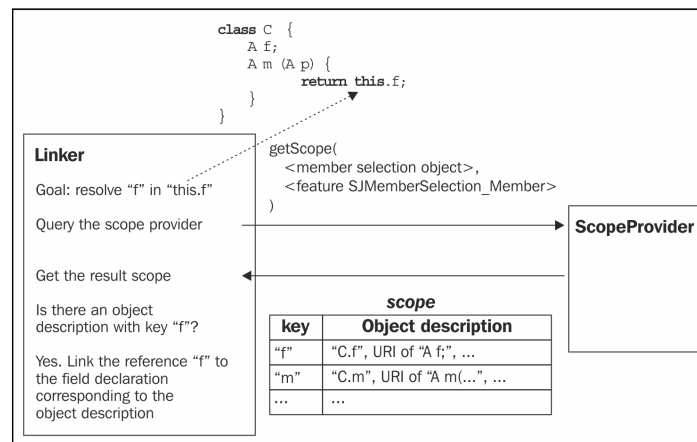
As we hinted, the outer scopes are consulted only if a matching name is not found in the current scope; this implements shadowing, which is typical of many DSLs.

For example, in the following SmallJava program:

```
class C {
  A m(A a) {
    A a = null;
    return a; // assume this is the context
  }
}
```

The local variable declaration a shadows the method parameter in the context of the return statement. This can be verified with the following test, which checks that the symbol reference a in the `return` statement's expression is linked to the variable declaration instead of the parameter with the same name:

```
@Test def void testVariableShadowsParamLinking() {
  '''
  class C {
    A m(A a) {
      A a = null;
      return a;
    }
  }
  class A {}
  '''.parse.classes.head.methods.head => [
      // the local variable should shadow the param
      body.statements.head.assertSame(
        (returnStatement.expression as SJSymbolRef).symbol)
    ]
}
```

Summarizing, the default implementation of the scope provider fits most DSLs; in fact, for the Entities DSL, the cross-reference resolution worked out of the box.
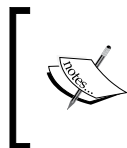
> The scope provider is also used by the content assist to provide a list of all visible elements. Since the scope concerns a specific program context, the proposals provided by the content assist are actually sensible for that specific context.

Component interaction

Before getting into scope customization, we conclude this section by summarizing, in a simplified way, how all of the preceding mechanisms are executed internally by Xtext:

- **Parsing**: Xtext parses the program files and builds the corresponding EMF models; during this stage, cross-references are not resolved

- **Indexing**: All the contents of the EMF models are processed; if an element can be given a qualified name, a corresponding object description is put in the index

- **Linking**: The linking service performs cross-reference resolution using the scope provider

This workflow is fixed. Consequently, we cannot rely on resolved cross-references during indexing. We need to keep that into consideration if we modify the indexing process (as we will see later in the section, *What to put in the index?*).

Custom scoping

As soon as a DSL introduces more involved features, such as nested blocks or inheritance relations, the scope provider must be customized according to the semantics of the language.

The Xtext generator generates an Xtend stub class for implementing a custom scope provider. In the SmallJava DSL it is `SmallJavaScopeProvider`.

One way of customizing the scope provider is to redefine the method `getScope`; this requires you to manually check the reference and the context's class, for example:

```
override getScope(EObject context, EReference reference) {
  if (reference == SmallJavaPackage::eINSTANCE.SJSymbolRef_Symbol) {
    if (context instanceof ...)
      ...
  } else if (reference ==
        SmallJavaPackage::eINSTANCE.SJMemberSelection_Member) {
    ...
  } else {
    super.getScope(context, reference)
  }
}
```

Xtext provides an easier declarative way of customizing the scope that reflects the mechanisms for customizations as we have seen in *Chapter 6*, *Customizations*, using a convention on method names and their parameters. In particular, you can declare methods in your scope provider according to one of the following patterns:

```
IScope scope_<EClass name>_<EFeature name>(<Context type> context,
                                           EReference reference)
IScope scope_<Object type>(<Context type> context,
                           EReference reference)
```

With the first pattern we declare that we want to provide a custom scope for the feature `<EFeature name>` of the class `<EClass name>` when the context is of type `<Context type>`. For example, if we wanted to provide a custom scope for the feature `symbol` of the class `SJSymbolRef` in the context of an `SJIfStatement` object, we would write the following method:

```
def scope_SJSymbolRef_symbol(SJIfStatement context, EReference r)
```

This first pattern allows us to be as specific as possible concerning the reference. The second pattern can be used when we want to provide a custom scope for a given element type and that scope can fit all cross-references of that type. In particular, if a specific type is referred only in a single rule, we do not need to be specific on the feature name. For example, in SmallJava, `SJSymbol` can be referred only in the rule `SJSymbolRef`, thus, we could have achieved the same goal by writing the following method:

```
def scope_SJSymbol(SJIfStatement context, EReference r)
```

Xtext will search for such methods reflectively when it has to build a scope. If there is no such method matching the actual context type, Xtext will repeat the search using the container of the context. It will keep on walking the containment hierarchy until it finds a matching method. This allows us to reuse the same scope computation in different places of the containment hierarchy. It is up to the developer to understand which context is required to be able to build the scope. Usually, when writing these methods in the custom scope provider, the context type corresponds to the `EClass` name (this holds for SmallJava, as we will see in the following sections).

> Since these methods are searched by Xtext using reflection, there is no check concerning the correctness of the method name. If you misspell the feature or the class name in the method name, that method will simply never be called; thus, you must pay attention when writing the signature of these methods.

Scope for blocks

The default scope provider can be too permissive. For example, in SmallJava, it allows an expression to refer to a variable declaration that is defined after that expression. As we have seen in *Chapter 8, An Expression Language*, forward references for variables are usually not permitted in languages. Moreover, if the language has nested code blocks, the default scope provider allows an outer block to access variable declarations of an inner block.

Let us consider an example of nested blocks in SmallJava:

```
class C {
  A m(A p) {
    A v1 = v4; // forward reference
    if (true) {
      A v2 = null;
      A v3 = v4; // forward reference
    }
    A v4 = v2; // reference to a var of an inner block
    return null;
  }
}
```

The example shows variable references that should not be valid. However, with the default scope provider, all variable declarations are visible in all contexts of the method body.

In *Chapter 8, An Expression Language*, we solved the problem of forward references by implementinga check in the validator, while in this chapter we show how to solve the visibility of symbols with a custom scope. The idea of the solution is basically the same: we want to build a scope consisting of the list of symbols defined before the current program context; while in *Chapter 8, An Expression Language*, we implicitly had one single block. In SmallJava we have nested blocks, and we also need to examine containing blocks. Moreover, when examining the containing block, we still need to limit the list of visible variables to those declared before the block containing the current context. This implicitly prevents a variable definition from referring to itself in its initialization expression. Considering the previous example, when the context is the variable declaration A v3 = v4; in the if block, the scope should consist of all the variables of the container (the if block) declared before the context, that is, only v2. The outer scope should consist of all the variables of the container of the container (the method's body) declared before the container of the context, that is, only v1, and so on. The end of this recursion will be the case when the container is the method itself, in which case the scope is the list of parameters. Thus, we want to build the following nested scope for the variable declaration A v3 = v4; in the if block:

```
v2, outer: (v1, outer: (p, outer: null))
```

To actually create the scope instance, which would require us to build a list of IEObjectDescription, we can use the static utility method Scopes.scopeFor. This method takes a list (actually an iterable) of EObject and creates a scope with the corresponding IEObjectDescription elements; in particular, the object descriptions are created using the feature name of the passed EObject elements. This way, the descriptions will have simple names instead of qualified ones, which is what we want. You can also pass the outer scope to the method Scopes.scopeFor.

To follow the convention of the scope provider method names, we write the following method:

```
def scope_SJSymbolRef_symbol(SJExpression context, EReference r)
```

As we said, in this DSL, we could have instead written:

```
def scope_SJSymbol(SJExpression context, EReference r)
```

We use a generic SmallJava expression as the context since we do not need other specific contextual information to build the scope according to the detailed strategy. The implementation is as follows:

```
def scope_SJSymbolRef_symbol(SJExpression context, EReference r) {
  context.eContainer.symbolsDefinedBefore(context)
}

def dispatch IScope symbolsDefinedBefore(EObject cont, EObject o){
  cont.eContainer.symbolsDefinedBefore(o.eContainer)
}

def dispatch IScope symbolsDefinedBefore(SJMethod m, EObject o) {
  Scopes::scopeFor(m.params) // this ends recursion
}

def dispatch IScope symbolsDefinedBefore(SJBlock b, EObject o) {
  Scopes::scopeFor(
    b.statements.variablesDeclaredBefore(o),
    b.eContainer.symbolsDefinedBefore(o.eContainer) // outer scope
  )
}

def private variablesDeclaredBefore
                (List<SJStatement> list, EObject o) {
  list.subList(0, list.indexOf(o)).
    filter(typeof(SJVariableDeclaration))
}
```

The implementation of the method `scope_SJSymbolRef_symbol` calls the recursive Xtend dispatch method `symbolsDefinedBefore`. The generic case of the dispatch method simply performs another step of the recursion, without participating in the building of the scope. The case for `SJMethod` ends the recursion by returning a scope consisting of the parameters of the method. The case for `SJBlock` builds a scope consisting of the variables in the block defined before the passed object; the outer scope is the one returned by the recursive invocation walking the containment hierarchy. Recall that the case for `SJBlock` includes the case for `SJMethodBody` and the case for `SJIfBlock`.

The following table illustrates the execution of the recursive dispatch method on the variable declaration `A v3 = v4`. The table shows the container, the context, and the symbols defined in the container before the context, that is, the scope for symbols in that context; the table also shows the matched dispatch method's parameters:

Container	Context	Symbols defined before the context	Matched dispatch method's parameters
If block	A v3 = v4	v2	(SJBlock, EObject)
If statement	If block		(EObject, EObject)
Method body	If statement	v1	(SJBlock, EObject)
Method definition	Method body	p	(SJMethod, EObject)

Note that when the context is `SJIfBlock`, the container is `SJIfStatement`, and there is no symbol to add in the scope.

The resulting scope is as we wanted:

```
v2, outer: (v1, outer: (p, outer: null))
```

In this way, we get the correct behavior for symbol references, which can be verified in the following test:

```
@Test def void testScopeProviderForSymbols() {
  '''
  class C {
    A m(A p) {
      A v1 = null;
      if (true) {
        A v2 = null;
        A v3 = null;
      }
      A v4 = null;
      return null;
    }
  }
  class A {}
  '''.parse.classes.head.methods.last.body.eAllContents.
  filter(typeof(SJVariableDeclaration)) => [
    findFirst[name == 'v3'].expression.assertScope
```

```
        (SmallJavaPackage::eINSTANCE.SJSymbolRef_Symbol,
           "v2, v1, p")
    findFirst[name == 'v4'].expression.assertScope
        (SmallJavaPackage::eINSTANCE.SJSymbolRef_Symbol,
           "v1, p")
   ]
 }
```

In this test, we verify the scope when the context is the variable declaration with name v3, and when the context is the variable declaration with name v4.

With this scope provider implementation, the variables defined in an inner block or after the program context cannot be referred; the user will get errors of the type **Couldn't resolve reference to...** as illustrated in the following screenshot:
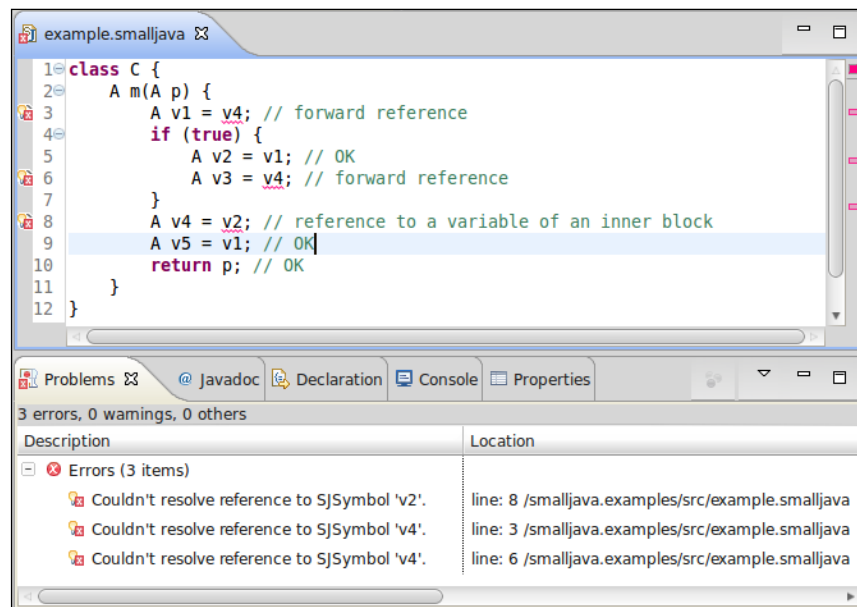
Scope for inheritance and member visibility

The default scope provider implementation based on containments does not work for more complex relations implied by the semantics of the DSL. A typical example is the inheritance relation in object-oriented languages.

In SmallJava a subclass' method body must be able to refer to superclass' members (note that for the moment, SmallJava does not have access level modifiers like `private` and `protected`: every member is implicitly `public`; we will add such modifiers in the next section).

For example, consider the following code snippet:

```
class C {
  A a;
  A n() { return null; }
}

class D extends C {
  A b;
  A m() {
      this.n(); // cannot access inherited method
      return this.a; // cannot access inherited field
  }
}
```

With the current scope provider, the members a and n of C will be visible only by their qualified name in the subclass D. Thus, the body of m of the subclass D is not able to refer to the members of the superclass C.

Moreover, the default scope provider implementation's strategy of considering only the containment relation cannot take into consideration the actual receiver of a member selection, which is instead needed for a valid member access.

For example, with the current scope provider implementation the following code would be considered valid:

```
class C {

}

class D {
  A b;
  A m() { return new C().b; }
}
```

While it should not be well-typed in SmallJava: the field b is not defined in the class C of the receiver new C().

We must build a custom scope for resolving an SJMember reference in an SJMemberSelection expression with the following strategy:

1. Get the type C of the receiver expression.
2. Return a nested scope consisting of:
    °   All the members in C
    °   Recursively as outer scope, all the members of the superclass of C

Note that in this case, to be able to build the scope, we need the context of type
SJMemberSelection. Since we need the type of the receiver; we define this method
in our scope provider as follows:

```
@Inject extension SmallJavaTypeProvider
def scope_SJMember(SJMemberSelection sel, EReference r) {
  var parentScope = IScope::NULLSCOPE
  val type = sel.receiver.typeFor

  if (type == null || type.isPrimitive)
    return parentScope
  for (c : type.classHierarchy.reverseView) {
    parentScope = Scopes::scopeFor(c.members, parentScope)
  }
  Scopes::scopeFor(type.members, parentScope)
}
```

Let us examine the preceding method. If the receiver has a primitive type, we return
an empty scope, since primitive types have no members. The iteration on the class
hierarchy of the receiver (obtained through the method SmallJavaModelUtil.
classHierarchy) must be performed starting from the top superclass. Thus we use
the utility method of the Xtend library reverseView, since the outer scopes must be
built first. The final scope will consist of all the members in the class of the receiver
and the outer scopes are built iterating over the inheritance hierarchy. If the class of the
receiver has no superclass, the scope will simply consist of the members of that class.

This implementation will also make field shadowing and method redefinitions work
out of the box; a member in a subclass will be considered before a homonymous
member in a superclass (since the latter will be in an outer scope).

> You should never return null as a scope; if you need to return
> an empty scope you should return IScope::NULLSCOPE.

With this implementation of the scope provider, the members of a superclass will
be visible in the subclass; thus, the first example of this section is now considered
valid. At the same time, only the members in the class hierarchy of the receiver
will be visible in a member selection expression; thus, the second example will
be considered invalid.

The following test verifies whether references to fields are resolved correctly
(a similar test can be written for methods), in particular, if fields in subclasses
have the precedence over homonymous fields in superclasses:

```
@Test def void testFieldScoping() {
  '''
  class A {
    D a;
    D b;
    D c;
  }

  class B extends A {
    D b;
  }

  class C extends B {
    D a;
    D m() { return this.a; }
    D n() { return this.b; }
    D p() { return this.c; }
  }
  class D {}
'''.parse.classes => [
    // a in this.a must refer to C.a
    get(2).fields.get(0).assertSame
      (get(2).methods.get(0).returnExpSel.member)
    // b in this.b must refer to B.b
    get(1).fields.get(0).assertSame
      (get(2).methods.get(1).returnExpSel.member)
    // c in this.c must refer to A.c
    get(0).fields.get(2).assertSame
      (get(2).methods.get(2).returnExpSel.member)
  ]
}

def private returnExpSel(SJMethod m) {
  m.returnStatement.expression as SJMemberSelection
}
```

This implementation still has a problem though, due to the fact that we allow a method and a field to have the same name (as in Java). With this implementation of scoping, methods and fields are mixed in the scope and they appear in the scope with their declaration order. Thus, when referring to a member, the first one will be linked irrespective of whether the expression is a field selection or a method invocation.

Consider the following example:

```
class C {
  A m;
  A m() {
    return this.m();
  }
}
```

The method invocation expression will wrongly refer to the field m.

Indeed, this test fails:

```
@Test def void testFieldsAndMethodsWithTheSameName() {
  '''
  class C {
    A f;
    A f() {
      return this.f(); // must refer to method f
    }
    A m() {
      return this.m; // must refer to field m
    }
    A m;
  }
  class A {}
  '''.parse.classes.head => [
    // must refer to method f()
    methods.head.assertSame(methods.head.returnExpSel.member)
    // must refer to field m
    fields.last.assertSame(methods.last.returnExpSel.member)
  ]
}
```

We must build the scope for member selection according to whether it is a method invocation or a field selection. We distinguish between a method invocation and a field selection using the feature `methodinvocation`:

```
def scope_SJMember(SJMemberSelection sel, EReference r) {
  var parentScope = IScope::NULLSCOPE
  val type = sel.receiver.typeFor

  if (type == null || type.isPrimitive)
    return parentScope
  for (c : type.classHierarchy.reverseView) {
    parentScope = Scopes::scopeFor(c.selectMembers(sel),
      parentScope)
  }
  Scopes::scopeFor(type.selectMembers(sel), parentScope)
}

def selectMembers(SJClass type, SJMemberSelection sel) {
  if (sel.methodinvocation)
    type.methods + type.fields
  else
    type.fields + type.methods
}
```

Now, the previous test succeeds.

Why do we return the other members, instead of returning only methods or fields according to the actual member selection kind? Because, without this strategy a program like the following:

```
class C {
  C f;
  C m() { return this.f(); }
}
```

would issue an error saying that the member `f` cannot be resolved, and this would not be informative; instead, with the preceding scoping implementation, `f` is resolved and our validator issues an error saying that we are trying to perform method invocation using a field (see *Chapter 9, Type Checking*); such an error is much better.

Visibility and accessibility

One important thing to understand about scoping is that it should deal with visibility, not necessarily with validity. For example, an element can be visible in a certain program context, but that context should not access it.

To illustrate this concept, we introduce in SmallJava the **access level modifiers** as in Java. To keep the example simple, we will not consider the Java package-private modifier, which is the default in Java.

Thus, we modify field and method declarations with the optional access level specification:

```
SJField:
access=SJAccessLevel? type=[SJClass] name=ID ';';
SJMethod:
access=SJAccessLevel? type=[SJClass] name=ID
  '(' (params+=SJParameter (',' params+=SJParameter)*)? ')'
  body=SJMethodBody ;
enum SJAccessLevel:
  PRIVATE='private' | PUBLIC='public' | PROTECTED='protected';
```

To express access level we use an **enum rule**; this will correspond to a Java enum. An enum rule always has an implicit default value, which corresponds to the first value. Thus, if no access level is specified in a SmallJava program, the value PRIVATE will be assumed.
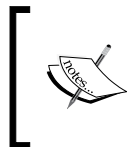
> Remember that, in Java, member access is checked at class level, not at object level. Inside a class C you can access private members on any object of class C, not only on this. We will use the same semantics in SmallJava.

To check if a member is accessible in a given program context, we need to check the subclass relation between the class containing the context and the class containing the referred member. We define this method in a dedicated class SmallJavaAccessibility as follows (we use the static methods of the class SmallJavaModelUtil):

```
def isAccessibleFrom(SJMember member, EObject context) {
  val contextClass = context.containingClass
  val memberClass = member.containingClass
  switch (contextClass) {
    case contextClass == memberClass : true
```

```
    case contextClass.isSubclassOf(memberClass) :
      member.access != SJAccessLevel::PRIVATE
    default:
      member.access == SJAccessLevel::PUBLIC
  }
}
```

This code should be self-explanatory: if the two classes are the same, the member can always be accessed. In a subclass you can access only members of superclasses that are not private; in all other cases, you can only access public members.

This test verifies our implementation for fields (a similar test is written for methods):

```
@Test def void testFieldAccessibility() {
  '''
  class A {
    private D priv;
    protected D prot;
    public D pub;
    D m() {
      this.priv; // private field
      this.prot; // protected field
      this.pub; // public field
      return null;
    }
  }

  class B extends A {
    D m() {
      this.priv; // private field
      this.prot; // protected field
      this.pub; // public field
      return null;
    }
  }

  class C {
    D m() {
      (new A()).priv; // private field
      (new A()).prot; // protected field
      (new A()).pub; // public field
      return null;
    }
  }
```

```
   class D {}
   '''.parse.classes => [
      // method in A
      get(0).methods.get(0).body.statements => [
        get(0).assertMemberAccessible(true)
        get(1).assertMemberAccessible(true)
        get(2).assertMemberAccessible(true)
      ]
      // method in B
      get(1).methods.get(0).body.statements => [
        get(0).assertMemberAccessible(false)
        get(1).assertMemberAccessible(true)
        get(2).assertMemberAccessible(true)
      ]
      // method in C
      get(2).methods.get(0).body.statements => [
        get(0).assertMemberAccessible(false)
        get(1).assertMemberAccessible(false)
        get(2).assertMemberAccessible(true)
      ]
   ]
}

def private assertMemberAccessible(SJStatement s,
                                   boolean expected) {
  val sel = s as SJMemberSelection
  expected.assertEquals(sel.member.isAccessibleFrom(sel))
}
```

Now we can modify the scope provider so that only the members that are accessible in a context are visible in that context. However, as we said at the beginning of the section, scoping should only deal with visibility, not with validity. In many ways, all members of a class are actually visible, the fact that some of them cannot be accessed is a validation issue.

Instead of restricting the scope, we write a validator rule that checks the accessibility of a referred member. With this strategy, we can provide better error information, for example, "private member is not accessible", instead of the default "couldn't resolve reference to...". In the editor, the user will still be able to navigate to the definition of a member even if the validator issues an error, since that member is still visible from the scoping point of view. This strategy is implemented by the Eclipse JDT editor as well, and it is considered a good practice (*Loose scoping, Strict validation*, see Zarnekow 2012).

The following is the validator rule:

```
class SmallJavaValidator extends AbstractSmallJavaValidator {
  @Inject extension SmallJavaAccessibility
  public static val MEMBER_NOT_ACCESSIBLE =
    "org.example.smalljava.MemberNotAccessible"
  @Check
  def void checkAccessibility(SJMemberSelection sel) {
    val member = sel.member
    if (member != null && !member.isAccessibleFrom(sel)) {
      error(
      '''The «member.access» member «member.name» is not accessible
here''',
        SmallJavaPackage::eINSTANCE.SJMemberSelection_Member,
        MEMBER_NOT_ACCESSIBLE
      )
  }
}
```

An example of accessibility error is shown in the next screenshot. In the subclass D, we can access the protected field f of superclass C, but not to the private method m (remember that members are private by default). However, m is correctly linked to its method definition, and the reported error clearly states the problem.
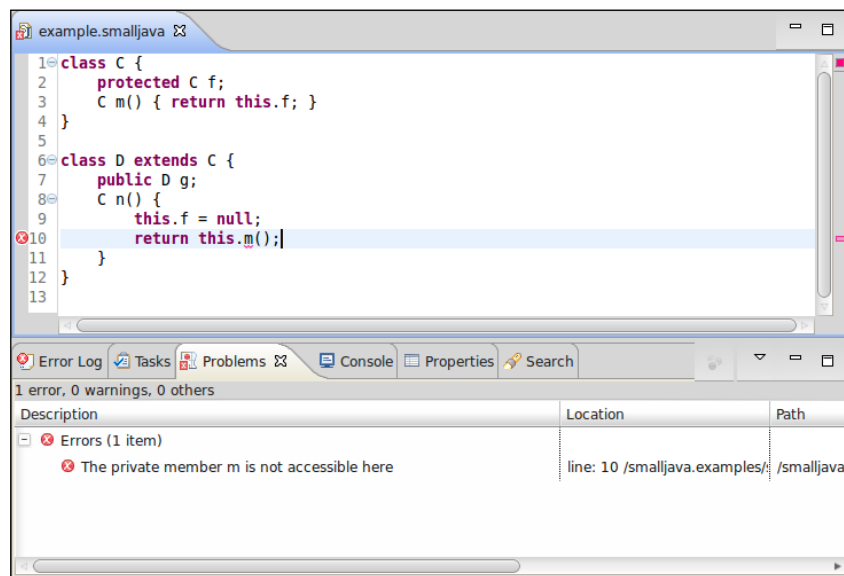
We adopted the same philosophy of a loose scoping and a strict validation in *Chapter 8*, *An Expression Language*, when checking forward references to variables. In this chapter, we rule out forward references to symbols with a restricted scope. In general, the validator approach allows you to give better feedback to the user, but it requires you to customize the content assist as shown in the following this section. On the contrary, the restricted scope approach does not require customization of other aspects, but the user receives errors of the type **Couldn't resolve reference to...**, which are less informative.

Filtering unwanted objects from the scope

Since we are not restricting the scope of the members, the content assist will still provide all members as proposals, even the private and the protected ones, irrespective of their accessibility in that specific program context. As we saw in *Chapter 8*, *An Expression Language*, it is good practice to modify the proposals so that only valid ones are presented to the programmer. In that chapter we manually built the proposals; in this chapter, since we introduced scoping, we use a different technique. The default implementation for proposals concerning a cross-reference relies on scoping; if you look at the implementation of the method `AbstractSmallJavaProposalProvider.completeSJSelectionExpression_Member`, you see that it uses the method `lookupCrossReference`, which in turn uses the scope provider. There is an overloaded version of `lookupCrossReference` that also takes a predicate to filter proposals. In Xtend, a predicate corresponds to a lambda returning a Boolean. Such a predicate receives `IEObjectDescription` as an argument and it returns true if that description has to be included in the proposals. We override `completeSJSelectionExpression_Member` in `SmallJavaProposalProvider` and we call `lookupCrossReference` passing a lambda to filter out members that are not accessible in that context:

```
class SmallJavaProposalProvider extends
            AbstractSmallJavaProposalProvider {
  @Inject extension SmallJavaAccessibility
  override void completeSJSelectionExpression_Member
            (EObject model, Assignment a,
             ContentAssistContext context,
             ICompletionProposalAcceptor acceptor) {
    lookupCrossReference
      (a.getTerminal() as CrossReference, context, acceptor) [
        description |
```

```
        (description.getEObjectOrProxy as SJMember)
          .isAccessibleFrom(model)
      ]
    }
  }
```

Note that in the lambda we retrieve `EObject` from the description.

> In the code of the SmallJava implementation, you can find some
> content assist tests for this customization.

In the next screenshot you can see that the content assist proposes only the members
that are actually accessible in the current context: the private method `m` of the
superclass is not proposed.
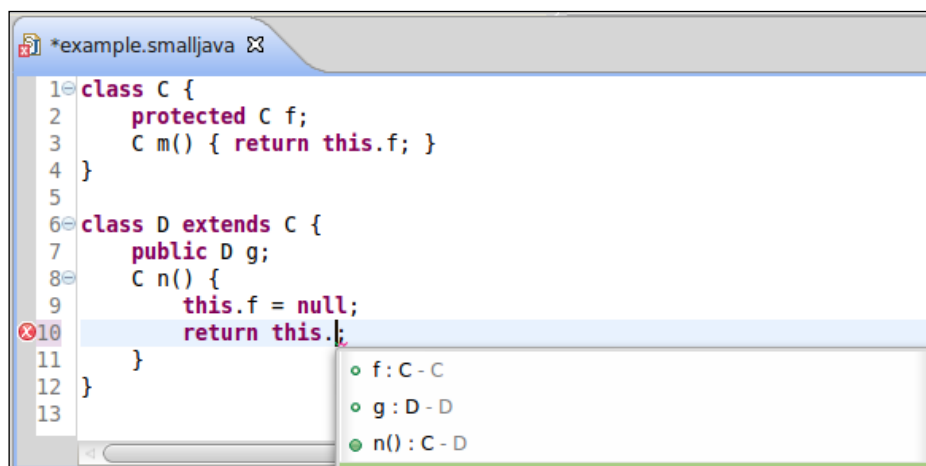
Global scoping

Xtext has a default mechanism for **global scoping** that allows you to refer to
elements defined in a different file, possibly in a different project of the workspace;
in particular, it uses the dependencies of the Eclipse projects. For Java projects, it uses
the classpath of the projects. Of course, this mechanism relies on the global index.

> Global scoping is implied by the fact that the default scoping
> mechanism always relies on an outer scope that consists of the
> visible object descriptions in the index.

With the default configuration in the MWE2, this mechanism for global scoping works out of the box. You can experiment with a project with some Entities DSL files. You can see, as illustrated in the following screenshot, that you can refer to the entities of another file; content assist works accordingly:
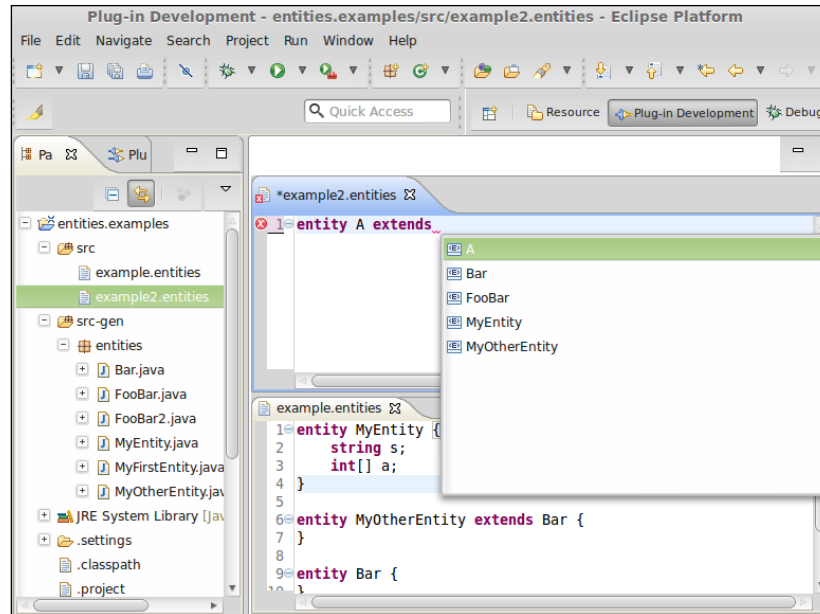
Before proceeding to the use of global scoping, it is worthwhile to learn how to write Junit tests that concern several input programs.

As hinted in the section *The index*, when running in a plain Java context where there is no workspace concept, the index is based on an EMF `ResourceSet`. When we have used the method `ParseHelper.parse` in our tests, we have always passed a string; there is an overloaded version of `parse` that also takes the resource set to be used when loading the passed program string. Thus, if we want to write a test that involves several files where one of them refers to elements of the other, we need to parse all input strings using the same resource set.

This can be accomplished in two ways. You can inject a `Provider<ResourceSet>`, create a resource set through this provider and pass it to the `parse` method as follows:

```
@Inject Provider<ResourceSet> resourceSetProvider;
@Test def void testTwoFiles() {
  val resourceSet = resourceSetProvider.get
  val first = '''class B extends A {}'''.parse(resourceSet)
  val second = '''class A { B b; }'''.parse(resourceSet)
  first.assertNoErrors
  second.assertNoErrors

  second.classes.head.assertSame(first.classes.head.superclass)
}
```

Note that in this test, the two input programs have mutual dependencies and the cross-reference mechanism works since we use the same resource set.

> It is crucial to validate the models only after all the programs are loaded; remember that `ParseHelper` only parses the program, it does not try to resolve cross-references.

Alternatively, you can parse the first input and retrieve its resource set from the returned model object; then, the subsequent inputs are parsed using that resource set:

```
@Test def void testTwoFilesAlternative() {
  val first = '''class B extends A {}'''.parse
  val second = '''class A { B b; } '''.
        parse(first.eResource.resourceSet)
   ... as before
```

In the following sections we will implement some aspects related to global scoping and the index.

Packages and imports

Since we can refer to elements of other files, it might be good to introduce the notion of namespace in SmallJava, which basically corresponds to the Java notion of a package.

Thus, we add an optional package declaration in the rule for `SJProgram`:

```
SJProgram:
  ('package' name=QualifiedName ';')?
  classes+=SJClass*;

QualifiedName: ID ('.' ID)* ;
```

The rule `QualifiedName` is a **data type rule**. A data type rule is like a terminal rule, for example, the terminal rule for `ID`, and it does not contain feature assignments. Differently from a terminal rule, a data type rule is valid only in specific contexts, that is, when it is used by another rule. This way, it will not conflict with terminal rules; for example, the rule `QualifiedName` will not conflict with the rule `ID`.

> The Xtext editor highlights a data type rule's name in blue.

According to the default mechanism for computing a qualified name (see the section *Qualified names*), when a `SJClass` is contained in a program with a package declaration, its fully qualified name will include the package name. For example, given this program:

```
package my.pack;
class C { }
```

the class `C` will be stored in the index with the qualified name `my.pack.C`.

It now makes sense to allow the user to refer to a SmallJava class with its fully qualified name, like in Java.

When you specify a cross-reference in an Xtext grammar, you can use the complete form `[<Type>|<Syntax>]`, where `<Syntax>` specifies the syntax for referring to the element of that type. The compact form `[<Type>]` we have used so far is just a shortcut for `[<Type>|ID]`. In fact, until now, we have always referred to elements by their `ID`. Now, we want to be able to refer to `SJClass` using the `QualifiedName` syntax, thus, we modify all the involved rules accordingly. We show the modified rule for `SJClass` (but also the rules for field, method, parameter, and variable declaration must be modified accordingly):

```
SJClass:
'class' name=ID ('extends' superclass=[SJClass|QualifiedName])?
'{' members += SJMember* '}' ;
```

Note that the rule for `QualifiedName` also accepts a single ID, thus, if there is no package, everything keeps on working as before.

We can now test class references with qualified names in separate files:

```
@Test def void testPackagesAndClassQualifiedNames() {
  val first = '''
  package my.first.pack;
  class B extends my.second.pack.A {}
  '''.parse
```

```
  val second = '''
package my.second.pack;
class A {
  my.first.pack.B b;
}
'''.parse(first.eResource.resourceSet)
first.assertNoErrors
second.assertNoErrors

second.classes.head.assertSame(first.classes.head.superclass)
}
```

Now it would be nice to have an **import** mechanism as in Java, so that we can import a class by its fully qualified name once and then refer to that class simply by its simple name. Similarly, it would be helpful to have an import with wildcard * in order to import all the classes of a specific package. Xtext supports imports, even with wildcards; it only requires that a feature with name importedNamespace is used in a parser rule and then the framework will automatically treat that value with the semantics of an import; it also handles wildcards as expected:

```
SJProgram:
  ('package' name=QualifiedName ';')?
  imports+=SJImport*
  classes+=SJClass*;

SJImport:
    'import' importedNamespace=QualifiedNameWithWildcard ';' ;
QualifiedNameWithWildcard: QualifiedName '.*'? ;
```

The following test verifies imports:

```
@Test def void testImports() {
  val first = '''
package my.first.pack;
class C1 { }
class C2 { }'''.parse

'''
package my.second.pack;
class D1 { }
class D2 { }'''.parse(first.eResource.resourceSet)

'''
package my.third.pack;
import my.first.pack.C1;
```

```
    import my.second.pack.*;

    class E extends C1 { // C1 is imported
      my.first.pack.C2 c; // C2 not imported
      D1 d1; // D1 imported by wildcard
      D2 d2; // D2 imported by wildcard
    }
    '''.parse(first.eResource.resourceSet).assertNoErrors
}
```

To keep the SmallJava DSL simple, we do not require the path of the `.smalljava` file to reflect the fully qualified name of the declared class as in Java. Indeed, in SmallJava, there is no concept of a `public` class.

The index and the containers

The index does not know anything about visibility across resources. Such a mechanism is delegated to `IContainer` that can be seen as an abstraction of the actual container of a given resource. The inner class `IContainer.Manager` provides information about the containers that are visible from a given container. Using these containers, we can retrieve all the object descriptions that are visible from a given resource.

> The implementation of the containers and the managers depends on the context of execution. In particular, when running in Eclipse, containers are based on Java projects. In this context, for an element to be referable, its resource must be on the classpath of the caller's Java project and it must be exported. This allows you to reuse for your DSL all the mechanisms of Eclipse projects, and the users will be able to define dependencies in the same way as they do when developing Java projects inside Eclipse. We refer to the section *About the index, containers, and their manager* of the Xtext documentation for all the details about available container implementations.

The procedure to get all the object descriptions which are visible from a given `EObject o` consists of the following steps:

1. Get the index.
2. Retrieve the resource description of the object `o`.
3. Use the `IContainer.Manager` instance to get all the containers in the index that are visible from the resource description of `o`.
4. Retrieve the object descriptions from the visible containers, possibly filtering them by type.

We thus add some utility methods to the class `SmallJavaIndex`:

```
class SmallJavaIndex {
  @Inject ResourceDescriptionsProvider rdp
  @Inject IContainer$Manager cm
  def getVisibleEObjectDescriptions(EObject o) {
    o.getVisibleContainers.map[ container |
      container.getExportedObjects
    ].flatten
  }

  def getVisibleEObjectDescriptions(EObject o, EClass type) {
    o.getVisibleContainers.map[ container |
      container.getExportedObjectsByType(type)
    ].flatten
  }

  def getVisibleClassesDescriptions(EObject o) {
    o.getVisibleEObjectDescriptions
      (SmallJavaPackage::eINSTANCE.SJClass)
  }

  def getVisibleContainers(EObject o) {
    val index = rdp.getResourceDescriptions(o.eResource)
    val rd = index.getResourceDescription(o.eResource.URI)
    cm.getVisibleContainers(rd, index)
  }...
```

Note that the result of `map` in the preceding code is a `List<Iterable<IEObjectDes cription>>`; the `flatten` utility method from the Xtend library combines multiple iterables into a single one. Thus, the final result will be an `Iterable<IEObjectDescription>`.

In the section *Exported objects*, we created the class `SmallJavaIndex` with utility methods to retrieve all the descriptions exported by a resource. We used those methods to write learning tests to get familiar with the index. The methods we have just added to `SmallJavaIndex` will be effectively used in the rest of the chapter to perform specific tasks that require access to all the visible elements.

Checking duplicates across files

The visibility of elements is implemented in the scope provider; thus, usually the index is not used directly. One of the scenarios where you must use the index is when you want to check for duplicates across files in a given container, that is, in a project and all its dependencies. The validator for SmallJava currently only implements checks for duplicates in a single resource.

We now write a validator method to check for duplicates across files using the index. We only need to check instances of `SJClass` since they are the only globally visible objects.
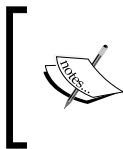
> Do not traverse the resources in the resource set (that is, visit all of them) since this is an expensive operation. Instead, use the index in these situations since this is both better and cheaper. It is OK to visit elements in the resource being processed.

The idea is to use the method `SmallJavaIndex.getVisibleClassesDescriptions` to get all the object descriptions of the type `SJClass` that are visible from the resource of a given SmallJava class and search for duplicate qualified names.

As shown in the section *Checking for duplicates* of *Chapter 9*, *Type Checking*, in order to check for duplicate elements, we search for an object that is different from the object being checked and that has the same name. When checking for duplicates across files using the index, we cannot rely on object identities completely since we only have object descriptions in the index and the corresponding object instances might not be loaded (after all, one of the aims of the index is to avoid loading all model objects). Thus, we also verify that the resource of the description is not the same as the resource of the currently processed class. We do that by checking the URI of the resources. Getting the URI of the resource of the class being processed is straightforward; to get the URI of the resource from the object description we get the URI of the object description and we trim the fragment part which refers to the position of the object in the resource:

```
@Inject extension SmallJavaIndex
@Inject extension IQualifiedNameProvider
public static val DUPLICATE_CLASS =
  "org.example.smalljava.DuplicateClass"
// perform this check only on file save
@Check(CheckType::NORMAL)
def checkDuplicateClassesInFiles(SJClass c) {
```

```
val className = c.fullyQualifiedName
c.getVisibleClassesDescriptions.forEach[
  desc |
  if (desc.qualifiedName == className &&
      desc.EObjectOrProxy != c &&
      desc.EObjectURI.trimFragment != c.eResource.URI) {
    error(
      "The type " + c.name + " is already defined",
      SmallJavaPackage::eINSTANCE.SJClass_Name,
      DUPLICATE_CLASS)
    return
  }
]
}
```

Note that we specified the `CheckType::NORMAL` in the `@Check` annotation: this instructs Xtext to call this method only on file save, not during editing as it happens normally (the default is `CheckType::FAST`). This is a good choice since this check might require some time, and if executed while editing, it might reduce the editor performance. Eclipse JDT also checks for class duplicates across files only on file save.

The following screenshot shows a project with two SmallJava programs with duplicate classes:
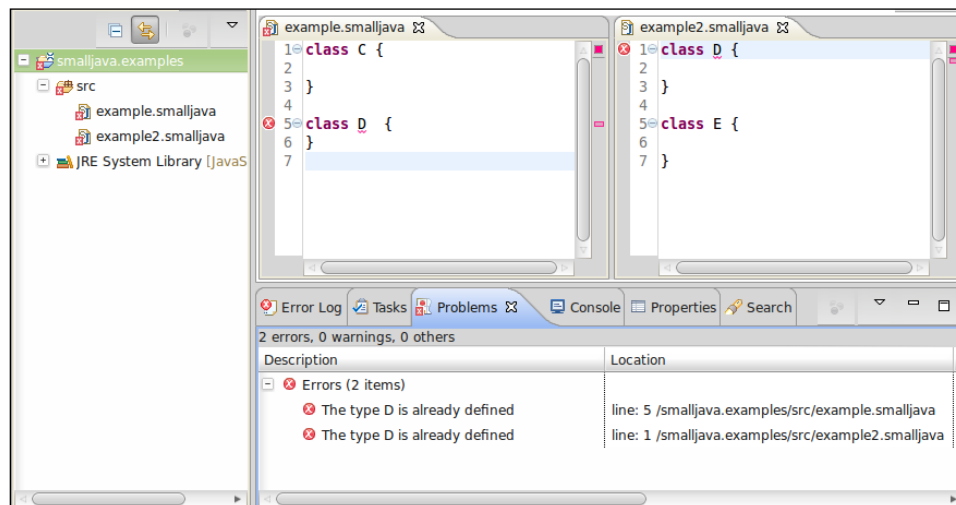
Providing a library

Our implementation of SmallJava does not yet allow us to make references to types such as `Object`, `String`, `Integer`, and so on. These types would allow us to declare variables initialized with constant expressions. In this section we show how to create a library with predefined types.

One might be tempted to hardcode these classes/types directly in the grammar, but this is not the best approach. There are many reasons for not doing that; mostly, that the grammar should deal with syntax only. Moreover, if we hardcoded, for example, `Object` in the grammar, we would only be able to use it as a type, but what if we wanted `Object` to have some methods? We would not be able to express that in the grammar.

Instead, we will follow the **library approach** (see also Zarnekow 2012-b). Our language implementation will provide a library with some classes, for example, `Object`, `String`, and so on, just like Java does. Since Xtext deals with EMF models, this library could consist of any EMF model. However, we can write this library just like any other SmallJava program.

To keep things simple, we write one single file, `mainlib.smalljava`, with the following classes:

```
package smalljava.lang;
class Object {
  public Object clone() {
    return this;
  }

  public String toString() {
    // fake implementation
    return "not implemented";
  }

  public Boolean equals(Object o) {
    // fake implementation
    return false;
  }
}

class String extends Object {}
class Integer extends Object {}
class Boolean extends Object {}
```

SmallJava does not aim at being usable and useful thus, this is just an example implementation of the classes of SmallJava library. We also use a package name, `smalljava.lang`, which reminds us of the main Java library, `java.lang`. We create this file in the source folder in the path `smalljava/lang/mainlib.smalljava`; furthermore, in the MANIFEST of the runtime SmallJava plug-in project we make sure that the package `smalljava.lang` is exported.

Now, if we run Eclipse, create a project, and add as dependency our `org.example.smalljava` project, the classes of `mainlib.smalljava` will be automatically available. In fact, Xtext global scoping implementation takes into consideration the project's dependencies; thus, the classes of our library are indexed and available in SmallJava programs. In the following screenshot you can see a plug-in project which depends on `org.example.smalljava`; the SmallJava files in this example project can then refer to the SmallJava classes defined in our library:
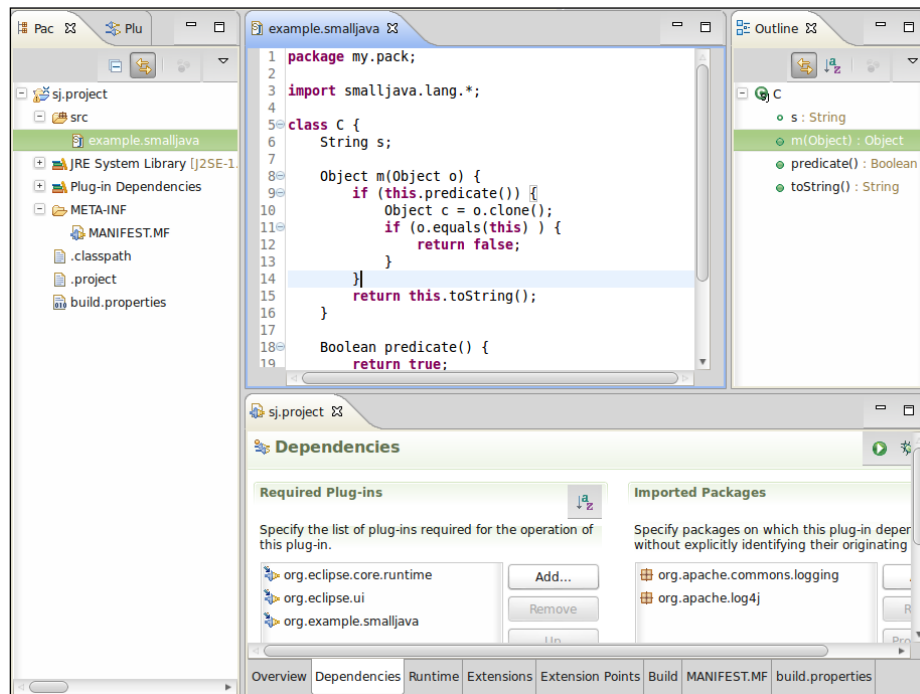
The implementation of SmallJava comes with a project wizard that creates a plugin project, adds the Xtext nature, adds the project `org.example.smalljava` as dependency, and creates an `example.smalljava` file with some contents. This wizard was automatically created by Xtext (and then customized) after enabling the corresponding fragment in the MWE2 file. We refer to the Xtext documentation for that.

Default imports

As we saw in the last sections, a DSL can automatically refer to elements in other files thanks to global scoping. In particular, Xtext also takes imported namespaces into consideration; if we import `smalljava.lang.*`, then we can refer to, for example, `Object` directly, without its fully qualified name. The scope provider delegates this mechanism to the class `ImportedNamespaceAwareLocalScopeProvider`.

At this point, in order to use library classes like `Object`, we have to explicitly import `smalljava.lang`. In Java you do not need to import `java.lang` since that is implicitly imported in all Java programs. It would be nice to implement this implicit import mechanism also in SmallJava for the package `smalljava.lang`. All we need to do is to provide a custom implementation of `ImportedNamespaceAwareLocalScopeProvider` and redefine the method `getImplicitImports` (the technical details should be straightforward):

```
class SmallJavaImportedNamespaceAwareLocalScopeProvider
        extends ImportedNamespaceAwareLocalScopeProvider {
  override getImplicitImports(boolean ignoreCase) {
    newArrayList(new ImportNormalizer(
    QualifiedName::create("smalljava", "lang"), true, ignoreCase
    ))
  }
}
```

Now, we need to bind this implementation in the runtime module. This is slightly different from other customizations in the Guice module we have seen so far; in fact, we need to bind the delegate field in the scope provider:

```
public class SmallJavaRuntimeModule extends
    AbstractSmallJavaRuntimeModule {
  @Override
  public void configureIScopeProviderDelegate(Binder binder) {
    binder.bind(org.eclipse.xtext.scoping.IScopeProvider.class)
    .annotatedWith(
      com.google.inject.name.Names
      .named(AbstractDeclarativeScopeProvider.NAMED_DELEGATE))
    .to(SmallJavaImportedNamespaceAwareLocalScopeProvider.class);
  }...
```

With this modification, we can simply remove the import statement `import smalljava.lang.*` and still be able to refer to the classes of the SmallJava library.

Using the library outside Eclipse

Being able to load the SmallJava library outside Eclipse is important both for testing and for implementing a standalone command-line compiler for the DSL.

As we saw in the previous sections, when we write unit tests with several dependent input programs, we need to load all the resources corresponding to input programs into the same resource set. Thus, we must load our library into the resource set as well to make the library available when running outside Eclipse.

We write a reusable class `SmallJavaLib`, which deals with all the aspects concerning the SmallJava library. In particular, we start with a method that creates the resource set, loads the library, and returns the resource set:

```
class SmallJavaLib {
  @Inject Provider<ResourceSet> resourceSetProvider;
  public val static MAIN_LIB = "smalljava/lang/mainlib.smalljava"
  def loadLib() {
    val stream =
      getClass().getClassLoader().getResourceAsStream(MAIN_LIB)
    resourceSetProvider.get() => [
      resourceSet |
      val resource =
        resourceSet.createResource(URI::createURI(MAIN_LIB))
      resource.load(stream, resourceSet.getLoadOptions())
    ]
}
```

The important thing here is that we get the contents of `mainlib.smalljava` by using the class loader; in particular, we use `getResourceAsStream`, which returns an `InputStream` to read the contents of the requested file. The class loader will automatically search for the given file using the classpath; this will work both for Junit tests and even when the program is bundled in a JAR as for the case of the standalone compiler. Then we create an EMF resource and load it using the contents of the library file. Note that an EMF resource does not necessarily have to correspond to a physical file; after all, in all the Junit tests we wrote so far, the `ParseHelper` has always created in-memory EMF resources with the string contents we passed to the `parse` method.

We are now able to write some tests to verify that implicit imports work correctly and also that the library itself contains no error. We use the version of the method `parse` that also takes the resource set as an argument; in particular, we use the resource set returned by the method `SmallJavaLib.loadLib` shown previously:

```
@RunWith(typeof(XtextRunner))
@InjectWith(typeof(SmallJavaInjectorProvider))
class SmallJavaLibTest {
  @Inject extension ParseHelper<SJProgram>
  @Inject extension ValidationTestHelper
  @Inject extension SmallJavaLib
  @Test def void testImplicitImports() {
  '''
  class C extends Object {
    String s;
    Object m(Object o) { return o.toString(); }
  }
  '''.loadLibAndParse.assertNoErrors
  }

  @Test def void testLibHasNoError() {
    loadLibrary
  }

  def private loadLibAndParse(CharSequence p) {
    p.parse(loadLibrary)
  }

  def private loadLibrary() {
    loadLib => [
      resources.forEach [contents.get(0).assertNoErrors]
    ]
  }
}
```

We also use `SmallJavaLib` to implement a standalone command-line compiler (see section *Standalone command-line compiler* of *Chapter 5*, *Code Generation*).

In the MWE2 workflow, we enable the following fragment:

```
// generator API
fragment = generator.GeneratorFragment {
    generateXtendMain = true
}
```

Since in SmallJava we did not implement a code generator, the compiler will only check that there are no errors in the input programs. We modify the generated `Main` Xtend class in order to load the library, load all the passed input files, and then validate all the resources in the resource set. Remember that we must validate the resources only after all the resources have been loaded, otherwise the cross-reference resolution will fail:

```
class Main {
  def static main(String[] args) {
    val injector =
      new SmallJavaStandaloneSetupGenerated().
        createInjectorAndDoEMFRegistration
    val main = injector.getInstance(typeof(Main))
    main.runGenerator(args)
  }

  @Inject IResourceValidator validator
  @Inject SmallJavaLib smallJavaLib

  def protected runGenerator(String[] strings) {
    // load the library
    val set = smallJavaLib.loadLib
    // load the input files
    strings.forEach[s | set.getResource(URI::createURI(s), true)]
    // validate the resources
    var ok = true
    for (resource : set.resources) {
      println("Checking " + resource.URI + "...")
      val issues = validator.
        validate(resource, CheckMode::ALL,
          CancelIndicator::NullImpl)
      if (!issues.isEmpty()) {
        for (issue : issues) {
          System::err.println(issue)
        }
        ok = false
      }
    }
    if (ok)
      System::out.println('Programs well-typed.')
  }
}
```

You can now follow the same procedure illustrated in *Chapter 5, Code Generation*, to export a runnable JAR file together with all its dependencies; the file `mainlib.smalljava` will be bundled in the JAR, and the class loader will be able to load it.

Using the library in the type system and scoping

Now that we have a library, we must update the type system and scope provider implementations in order to use the classes of the library. We declare public constants in `SmallJavaLib` for the names of the classes declared in our library:

```
class SmallJavaLib {
  public val static LIB_PACKAGE = "smalljava.lang"
  public val static LIB_OBJECT = LIB_PACKAGE+".Object"
  public val static LIB_STRING = LIB_PACKAGE+".String"
  public val static LIB_INTEGER = LIB_PACKAGE+".Integer"
  public val static LIB_BOOLEAN = LIB_PACKAGE+".Boolean"
```

We use these constants to modify `SmallJavaTypeConformance` to define special cases as follows:

```
class SmallJavaTypeConformance {
  @Inject extension IQualifiedNameProvider
  def isConformant(SJClass c1, SJClass c2) {
    c1 == nullType || // null can be assigned to everything
    (conformToLibraryTypes(c1, c2)) ||
    c1 == c2 ||
    c2.fullyQualifiedName.toString == SmallJavaLib::LIB_OBJECT ||
    c1.isSubclassOf(c2)
  }

  def conformToLibraryTypes(SJClass c1, SJClass c2) {
    (c1.conformsToString && c2.conformsToString) ||
    (c1.conformsToInt && c2.conformsToInt) ||
    (c1.conformsToBoolean && c2.conformsToBoolean)
  }

  def conformsToString(SJClass c) {
    c == stringType ||
    c.fullyQualifiedName.toString == SmallJavaLib::LIB_STRING
  }... similar implementations for int and boolean
```

Concerning the conformance for library types, the type of string constant expression is conformant to the library class `String`. The cases for Boolean and integer expressions are similar. Each class is considered conformant to the library class `Object` (as in Java), even if that class does not explicitly extend `Object`. If we introduce basic types directly in SmallJava, for example, `int` and `boolean`, we would still have to check conformance with the corresponding library classes, for example, `Integer` and `Boolean`.

The fact that every SmallJava class implicitly extends the library class `Object` must be reflected in the scope provider, so that any class is able to access the methods implicitly inherited from `Object`. For example, the class should be well-typed even if it does not explicitly extend `Object`, but the current implementation rejects it, since it cannot resolve the references to members of `Object`:

```
class C {
  Object m(Object o) {
    Object c = this.clone();
    return this.toString();
  }
}
```

To solve this problem, we first add a method in `SmallJavaLib` that loads the EMF model corresponding to the library `Object` class:

```
class SmallJavaLib {
  @Inject extension SmallJavaIndex
  ...

  def getSmallJavaObjectClass(EObject context) {
    val desc = context.getVisibleClassesDescriptions.findFirst[
        qualifiedName.toString == LIB_OBJECT]
    if (desc == null)
      return null
    var o = desc.EObjectOrProxy
    if (o.eIsProxy)
      o = context.eResource.resourceSet.
        getEObject(desc.EObjectURI, true)
    o as SJClass
  }
```

In the preceding code , we get the object description of the library class `Object` using `SmallJavaIndex.getVisibleClassesDescriptions`. If the `EObject` corresponding to `Object` is still a proxy, we explicitly load the actual `EObject` from the resource set of the passed context using the URI of the object description. The preceding code assumes that the library classes are visible in the current projects. It also assumes that the `EObject context` is already loaded, otherwise the resolution of the proxy will fail.

Now, we can implement a method that returns the class hierarchy of a SmallJava class with the library `Object` class added at the end of the hierarchy. If the class explicitly extends `Object`, we do not need to modify the hierarchy:

```
class SmallJavaLib {
  @Inject extension IQualifiedNameProvider
  ...

  def getClassHierarchyWithObject(SJClass c) {
    var hierarchy = c.classHierarchy
    if (hierarchy.last?.fullyQualifiedName?.toString
        != LIB_OBJECT) {
      val smallJavaObjectClass = getSmallJavaObjectClass(c)
      if (smallJavaObjectClass != null)
        hierarchy += smallJavaObjectClass
    }
    hierarchy
  }
}
```

Finally, we modify our scope provider so that it uses this method to get the class hierarchy and build the scope for members; this will transparently make the members from `Object` visible to any class and requires a very small modification to the scope provider:

```
class SmallJavaScopeProvider extends AbstractDeclarativeScopeProvider
{
  @Inject extension SmallJavaLib
  def scope_SJMember(SJMemberSelection sel, EReference r) {
    var parentScope = IScope::NULLSCOPE
    val type = sel.receiver.typeFor
    if (type == null || type.isPrimitive)
        return parentScope
    for (c : type.classHierarchyWithObject.reverseView) {
      ...as before
```

Note that it is important to make it possible to easily modify the structure of the library classes in the future. We did hardcode as public constants the fully qualified names of library classes in `SmallJavaLib`, but not the library classes' structure, so if in the future we want to modify the implementation of the library classes, we will not have to modify the type system neither the scope provider.

The other interesting feature is that one could easily provide a different implementation of the library as long as the main library class names are kept. The current SmallJava implementation will seamlessly be able to use the new library without any change. This is another advantage of keeping the DSL and the library implementations separate.

Dealing with super

As a final feature, we add the mechanism for invoking the implementation of a method in the superclass by using the keyword `super`. Note that `super` should be used only as the receiver of a member selection expression, that is, it cannot be passed as the argument of a method. Following the practice "loose grammar, strict validation", we do not impose this at the grammar level. Thus, we add the rule for `super` as a terminal expression:

```
SJTerminalExpression returns SJExpression:
    ...
    {SJSuper} 'super' | ...
```

and we add a validator rule that checks the correct `super` usage:

```
@Check
def void checkSuper(SJSuper s) {
  if (s.eContainingFeature !=
        SmallJavaPackage::eINSTANCE.SJMemberSelection_Receiver)
    error("'super' can be used only as member selection receiver",
      null, WRONG_SUPER_USAGE)
}
```

Thanks to the way we implemented the scope provider, in order to make members of the superclass visible when the receiver expression is `super`, we only need to provide a type for `super`. The type for `super` is the superclass of the containing class:

```
class SmallJavaTypeProvider {
  def typeFor(SJExpression e) {
    switch (e) {
      SJThis : e.containingClass
      SJSuper : e.containingClass.superclass
      ...as before
```

However, this way, if a class does not explicitly extend `Object`, it will not be able to access the methods of `Object` with `super`. To solve this, we just need to return the loaded instance of `Object` in case the superclass is null using `SmallJavaLib.getSuperclassOrObject`:

```
class SmallJavaTypeProvider {
  @Inject extension SmallJavaLib
  def typeFor(SJExpression e) {
    switch (e) {
      SJThis : e.containingClass
      SJSuper : e.containingClass.getSuperclassOrObject
      ...as before
  }

  def getSuperclassOrObject(SJClass c) {
    c.superclass ?: getSmallJavaObjectClass(c)
  }
```

The following screenshot shows a usage example of `super`. Note that the subclass redefines the method `m` and changes the return type with a subtype (as we saw in the section *Checking method overriding* of *Chapter 9, Type Checking*, this is valid). You can see that `super.m()` refers to the implementation in the superclass; the superclass `m` returns an object of type `C`, thus we get an **Incompatible types** error.
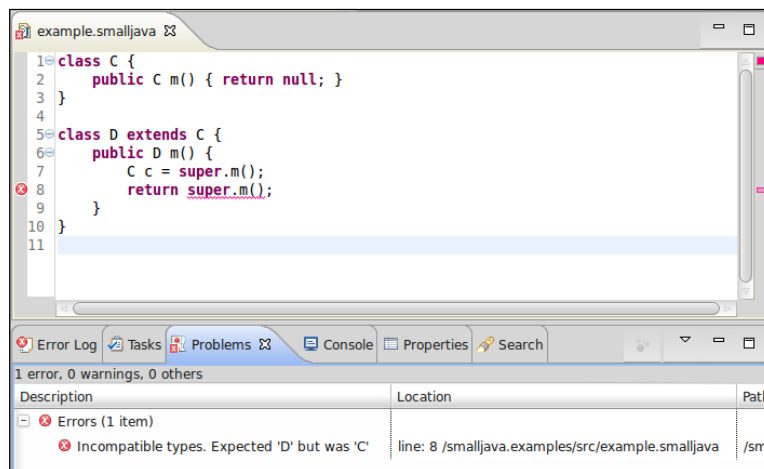
What to put in the index?

As explained earlier in this chapter, everything that can be given a name will have a corresponding entry in the index; moreover, by default, each element of the index can be referred through its fully qualified name. However, only the references that use the qualified name syntax can refer to these elements using the index. In SmallJava, only classes can be referred with qualified names.

Therefore, it makes no sense to index those elements that cannot be directly accessed through a qualified name. In our DSL, this means that only classes should be indexed. Although the presence of entries in the index for SmallJava methods, fields, and local variables does not harm, still it occupies some memory space uselessly. Moreover, the indexing procedure could be optimized by removing the overhead of indexing useless elements.

We then tweak the strategy for building the index for SmallJava programs; we need to provide a custom implementation of `DefaultResourceDescriptionsStrategy` and redefine the method `createEObjectDescriptions`. This method is automatically called by Xtext when the index is built or updated when resources change. We implement our custom version that creates object descriptions only for SmallJava classes:

```
@Singleton
class SmallJavaResourceDescriptionsStrategy extends
DefaultResourceDescriptionStrategy {
  @Inject extension IQualifiedNameProvider
  override createEObjectDescriptions(EObject eObject,
                IAcceptor<IEObjectDescription> acceptor) {
    if (eObject instanceof SJProgram) {
      (eObject as SJProgram).classes.forEach[
        sjClass |
        val fullyQualifiedName = sjClass.fullyQualifiedName
        if (fullyQualifiedName != null)
          acceptor.accept(
            EobjectDescription::create(fullyQualifiedName,
                sjClass)
          )
      ]

      true
    } else false
  }
}
```

Note that this class must be annotated with @Singleton, indicating that only one instance per injector will be used for all injections for this class.

Of course, we bind our implementation in the runtime module:

```
public class SmallJavaRuntimeModule extends
    AbstractSmallJavaRuntimeModule {
  public Class<? extends IDefaultResourceDescriptionStrategy>
              bindIDefaultResourceDescriptionStrategy() {
    return SmallJavaResourceDescriptionsStrategy.class;
  }...
```

This way, we improve the default indexing behavior.

> If you now run the test method testExportedEObjectDescriptions shown in the section *Exported objects,* you will see that it fails. You will need to modify it since the only descriptions in the index will be the ones of the classes C and A.

As we hinted at in the previous chapter, NamesAreUniqueValidator uses the index to quickly check for duplicate names of a given type. With our custom strategy, duplicate names would be detected for classes only. However, as we showed in the previous chapter, we disabled the NamesAreUniqueValidator for SmallJava and we implemented duplicate name checks manually.

> In general, if you modify the strategy for building the index, you should also manually customize the uniqueness check since NamesAreUniqueValidator will surely not work.

Additional automatic features

Xtext makes use of the index to automatically provide many additional IDE features for your DSL. Some examples are shown in the next screenshot. For example, you can mark occurrences of any named element by toggling the corresponding toolbar button. In the following screenshot, it is the one right on top of the **Plug-in Development** perspective button, marked with **(1)**. The markers will be evident both in the editor and in its right-hand side ruler. This feature is based on the IResourceDescription instances stored in the index; they contain information about cross-references, possibly to other resources. Furthermore, by right-clicking on an element in the editor, you can choose the menu **References**, and in the **Search** view, you can see all the files in your project that reference the selected element. In the following screenshot, we selected the method predicate in the file example.smalljava, and the view shows all its occurrences also in the other file example2.smalljava:
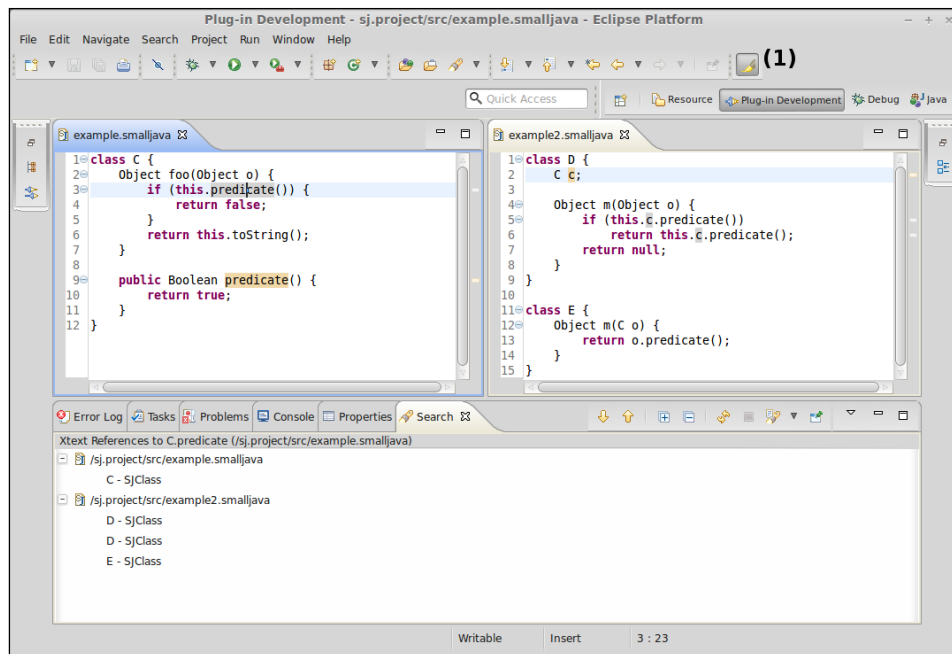
The editor for your DSL automatically supports refactoring of names: just select an element with a name, right-click on it, and choose **Rename Element**. This refactoring has the same user interface as the one of JDT; you can also access the refactoring dialog so that you can have a preview of what will be modified and possibly deselect some modifications. Of course, refactoring works across multiple files as well. For instance, in the following screenshot we change a SmallJava method's name.
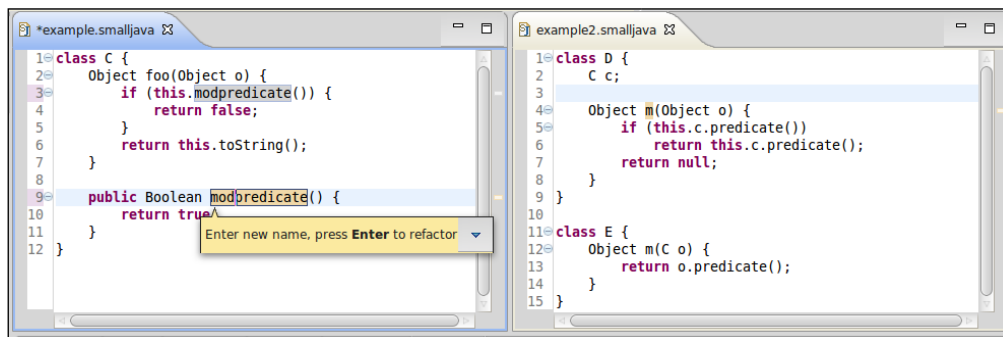
References in the same file are updated while you rename the element. After pressing *Enter*, you can note (see the following screenshot) that the method occurrences are modified accordingly in the other file.
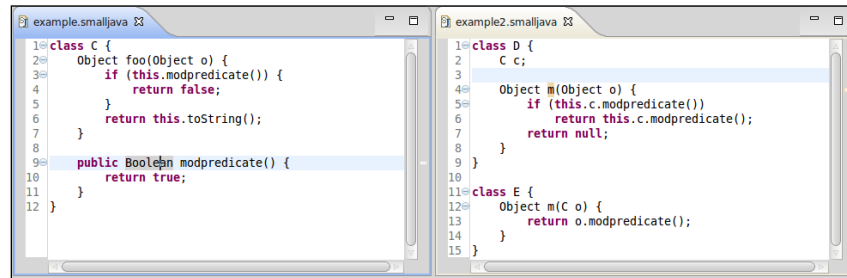
Summary

In this chapter we described scoping, which is the main mechanism behind visibility and cross-reference resolution. In particular, scoping and typing are often strictly connected and interdependent especially for object-oriented languages. We described both local and global scoping and we showed how to customize these mechanisms using the SmallJava DSL as a case study.

In the next chapter we will show how you can create a p2 repository for your DSL implementation; this way, other users can easily install it in Eclipse. Xtext provides a wizard that creates the entire infrastructure to build a p2 repository with Buckminster, an Eclipse project for automatic building.

11
Building and Releasing

In this chapter we describe how you can release your DSL implementation by creating an Eclipse p2 repository. In this way others can easily install it in Eclipse. With this respect, Xtext provides a wizard that creates the infrastructure to build a p2 repository with Buckminster, an Eclipse project for automatic building. The wizard will also create all the needed files to build your projects and test them in a headless way, that is, outside Eclipse. This makes it easy to run your builds on a continuous integration server.

Release engineering

Once your DSL implementation reaches a mature state, you would like to make it available on the Internet so that others can install it in their Eclipse. You could use the standard Eclipse **Export** wizard to create ZIP files that others can extract in their Eclipse installation. However, this installation method was deprecated many years ago, in favor of **p2 repositories** (also called **p2 sites**), which in turn, have replaced old style Eclipse update sites. Most of the sites you have been using to install new features into your Eclipse are p2 repositories, thus, you should build a p2 repository for your DSL, if you want users to easily install it.

In software engineering, **release engineering**, abbreviated as **releng**, concerns the compilation, assembly, and delivery of source code into finished products or other software components. In this section we briefly introduce some scenarios which require release engineering mechanisms. These will be connected to the creation of an installable version of your software and also to the capability to build and test your projects in isolation, outside Eclipse, in an automatic way.

Headless builds

Another important issue when developing a DSL (and in general, any project) is that you should be able to build all your projects **headlessly**, that is, from the command line outside Eclipse, in an automatic way. This will give you more confidence that your plug-ins can be installed in other Eclipse installations without problems. Installation problems easily go unnoticed, although it was possible to build the software locally. If dependencies of your components are not present in the user's environment, and you did not describe these dependencies, it will not be possible for others to install your software. While developing, these problems are easily detected. It simply will not build, but when installing, there are more considerations such as, will it install on all intended platforms, which Eclipse versions will it work with, and so on.

For all of the stated reasons, being able to build your software in isolation is a requirement in software production. This way, the headless building process will not rely on your Eclipse installation, but on a separate set of dependencies which are specified in a separate configuration.

Target platforms

A **target platform** is a set of features and plug-ins that your software, for example, your DSL implementation, depends on, for compiling, testing, and running.

By using a defined target platform, you can easily separate the tools that you need to develop your software from the actual dependencies that are required to compile and execute your software. The tools are usually not needed during the automatic building process, thus, the target platform contains only what is actually needed to build and test your software. This way, the compilation of your plug-ins will be decoupled from the Eclipse development environment. This also holds when you test your plug-ins both with Junit tests and when you run another Eclipse instance. The launch configuration will run the new Eclipse instance with the plug-ins specified in the target platform, and not the ones of your Eclipse development environment.

> If no specific target platform is specified, the target platform defaults to the current Eclipse installation, that is why, even if you never explicitly define target platforms, everything works anyway.

Defining a target platform allows you to compile and test your software against dependencies without installing them in your Eclipse development environment. Defining different target platforms also allows you to test your software against different Eclipse versions; for example, you can develop with Eclipse Juno, but your software will be compiled and run using a target platform with Eclipse Kepler plug-ins.

If you develop your software in a team, all developers should use the same target platform definition. They can develop using different Eclipse installations with possibly different installed tools, but the compilation and the testing will use the same defined target platform, ensuring consistency.

In Eclipse, a target platform can be defined by using a **target definition** file, which can be created with the corresponding wizard and edited with the corresponding editor. Once a target platform is defined, it can be set as the current target platform (using the target platform definition file editor, or using the Eclipse corresponding preference page). This process will resolve all the specified requirements, possibly downloading them from remote repositories, and installing them in a separate place (so that it will not conflict with your Eclipse installation). Since such target definition files are XML files, a target definition can be easily shared since it can be put in the SCM repository (such as Git) used by all the developers of the team. Each developer only needs to open that file and set it as the current target platform.

Continuous integration

If your software consists of many loosely coupled plug-ins, you probably have tests that test them in isolation; however, you should also have tests for the integration of all of them, to make sure that they are able to run altogether without problems in the final execution environment. In this book we implemented both plain Junit tests and plug-in Junit tests (see *Chapter 7*, *Testing*). Although we learned to use launch configurations and test suites to easily run all the tests, still, runtime tests are run separately from integration UI tests.

Thus, running all tests after every modification might be a burden for the programmer, decreasing the production cycle. Typically, you run tests that concern a specific task/modification/new feature that you are working on. However, single component modifications should also be tested when integrated in the whole application. With this respect, **Continuous Integration**, often abbreviated as **CI**, (Fowler, 2006) is the practice in which isolated changes are immediately tested in the complete code base. In this way, if a bug is introduced into the application by a single component, it can be easily and quickly identified and corrected. For this reason, it is crucial to have a build automation mechanism which relies on a headless building process. Usually, the actual build process is delegated to a specific software run on a dedicated server. A complete integration build can take several minutes and the developer can continue working while the server executes the build and check the result periodically.

Xtext provides a nice wizard to set up the configuration files to allow you to easily create a release of your DSL. Furthermore, these files will also allow you to run a complete automatic build headlessly, including execution of tests. This way, most of the set up for build automation and continuous integration is already done for you by this wizard. The resulting building infrastructure created by this wizard relies on the Eclipse tool Buckminster, that we will introduce in the next section.

Introduction to Buckminster

Buckminster (`http://www.eclipse.org/buckminster`) is a set of frameworks and tools for automating the building and assembling of components. It comes in two versions: one is to be installed in the IDE and the second one is to be used headlessly. A very nice feature of Buckminster is that it provides the same tools in both environments. This means that the IDE builds and headless builds are performed the same way; you do not have to deal with multiple ways of building and the potential differences between build technologies.

Buckminster resolves all the dependencies of the software components needed to build, including those that are built from source. When Buckminster knows what is needed, it materializes them by fetching the content, creates workspace projects for source that should be built, and places everything else in the target platform. Buckminster supports many different mechanisms for both dependency resolution and materialization. You can work with the technologies you are already using, for example, the local filesystem, git, svn, maven repositories, and so on. You will not have to maintain two different project structures, one for the IDE and one for the headless build (for example, to be run on a continuous integration server). This is also a very striking and interesting feature of Buckminster with respect to other solutions such as, Eclipse PDE-Build (based on ANT scripts) and, most of all, Maven/Tycho. Note that maintaining two building infrastructures can really be a burden, especially when things will start working in the IDE but not in the headless build and vice-versa; this is especially true for unit tests. With Buckminster, you usually do not experience bad surprises in your headless builds since the building and testing infrastructure is the same.

Recently, the Tycho project (a set of Maven plug-ins and extensions for building Eclipse plug-ins with Maven, a build manager for Java projects) is receiving much attention in the Eclipse community for building and testing of Eclipse projects. Although Tycho fills most of the gap between Maven building mechanisms and Eclipse, still it is actually a different building system; you will still need to maintain two building structures and, very often, specify or repeat project dependencies into different places. Moreover, Junit tests during a headless Tycho build will not run exactly the same way as in the IDE.

Setting up a Tycho build for an Xtext project may not be easy, and you should get ready to spend some time to make things work in the IDE the same way as in the headless build and vice-versa. Xtext provides a wizard to easily set up all the configuration files for building with Buckminster. In spite of all the attention that Tycho is receiving, we strongly encourage you to get familiar with Buckminster to enjoy a nicer and easier building experience.

Installing Buckminster

To install Buckminster in your Eclipse, you need to use the repository that corresponds to your Eclipse version, for example:

- `http://download.eclipse.org/tools/buckminster/updates-4.2` for Eclipse **Juno**

- `http://download.eclipse.org/tools/buckminster/updates-4.3` for Eclipse **Kepler**

The two main features to install are: Buckminster – Core and Buckminster – PDE Support (these will be enough to create a p2 repository).

The installation of the headless version of Buckminster, which is required if you plan to build and test your software headlessly, possibly in a continuous integration server, requires some additional steps, as shown at `http://www.eclipse.org/buckminster/downloads.html`. However, the Xtext Buckminster wizard can install Buckminster headless for you, so you can rely on that. Moreover, in the sources of the examples of this book, you will find an ANT target that automatically installs Buckminster headless.

Using the Xtext Buckminster wizard

In this chapter we will use a brand new example DSL project to demonstrate the building and releasing mechanisms (of course, the same mechanisms can be applied to an existing DSL as well):

1. Go to **File | New | Project...**, in the dialog, navigate to the **Xtext** category and select **Xtext Project**.

2. In the next dialog you should specify the following names:

- **Project name**: `org.example.build.hello`
- **Name**: `org.example.build.hello.Hello`
- **Extensions**: `hello`
- Check the option **Create SDK feature project** (this option must be checked for this example to work)

The wizard will create four projects in the workspace and it will open the file `Hello.xtext`.

We will use this example only for building and releasing, thus, we are not interested in the DSL itself; we can simply leave the default grammar as it is. However, make sure to run the MWE2 workflow at least once, so that all the Xtext artifacts are generated.

We add a Junit Xtend test class, `HelloParserTest`, in the `org.example.build.hello.tests` project with two test methods, just to have some tests to run during the headless build.

Note that, different from the other examples, this time we also generate an SDK feature project. A feature project is required if you want your software to be installable via a p2 repository. You can only install features, not single plug-ins, with the Eclipse **Install New Software** wizard.

We now use the Xtext Buckminster wizard. Right-click on the `org.example.build.hello.sdk` project, and go to **New | Other...**. Expand the **Xtext** category, then the **Continuous Integration** category and select **Build with Buckminster** as shown in the following screenshot:
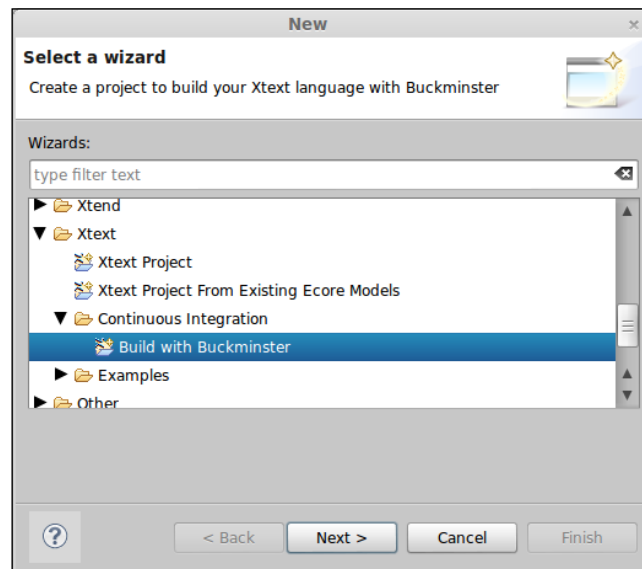
Let us now examine the wizard page (refer to the next screenshot). The wizard will inform you that it will create two additional projects:

- `org.example.build.hello.buckminster`, the releng project, that is, the project with all the files for the headless build
- `org.example.build.hello.site`, the project that will be used to create the p2 site

The wizard dialog accepts the path where the Buckminster headless installation should be found in your computer. It is not strictly required to specify this path now; the path is required only if you plan to build your DSL headless from the command line or in a continuous integration system. The wizard offers you to install Buckminster headless right at this time; you just need to click on the hyperlink in the wizard page. You can specify some launch configurations for Junit tests; these will be run during the headless build. The main Xtext project wizard already creates a launch configuration for Junit tests (we described this in *Chapter 7, Testing*), thus, we select `org.example.build.hello.tests.launch` in the `org.example.build.hello.tests` project. When you press **Finish**, the two additional projects will be created:
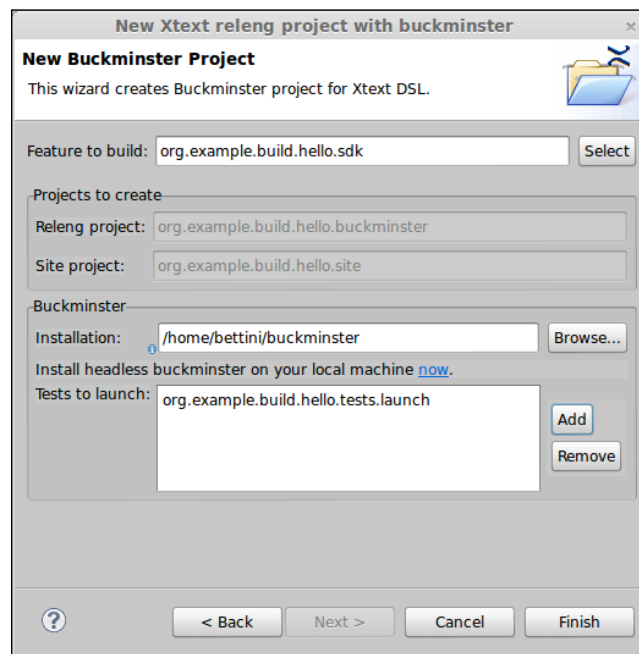
Building the p2 repository from Eclipse

We are now ready to create a p2 repository for our example DSL. We will do this from Eclipse by relying on Eclipse Buckminster.

To build a p2 repository, right click on the `org.example.build.hello.site` project and go to **Buckminster | Invoke Action...**; Buckminster offers many default actions for Eclipse plug-in and feature projects. The one we are interested in is **site.p2** (see the following screenshot); thus, select that action and press **OK**.
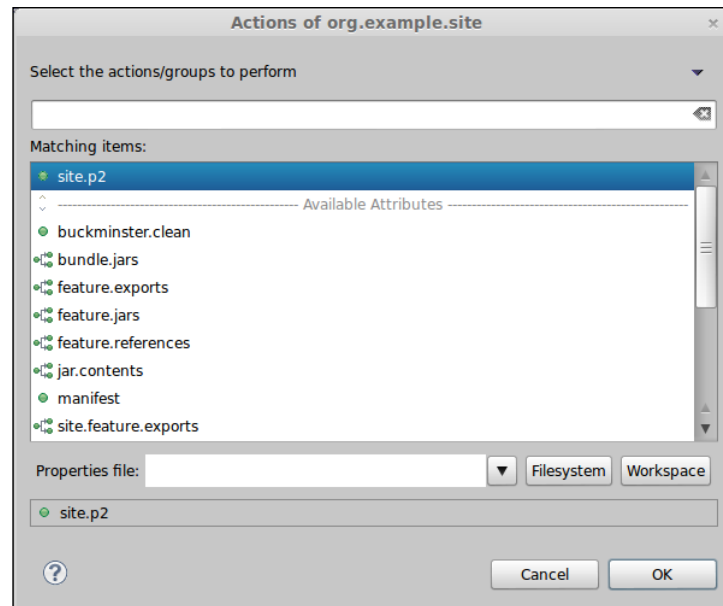
You will now see a dialog showing the progress and some output in the "Console" view. The procedure should terminate without errors.

By default, Buckminster will generate all its artifacts in the system temporary directory under `buckminster/build`. The p2 repository can then be found in the subdirectory `org.example.site_1.0.0-eclipse.feature/site.p2`.

You can change this path and many other properties used by Buckminster by providing a properties file. For example, we can create a file, say `buckminster.properties`, and store it in the `org.example.build.hello.site` project with the following contents:

```
# where all the output should go
buckminster.output.root=${user.home}/build/hello
```

This means that we want Buckminster to generate its output starting from the base path `build/hello` in our home directory.

We can now call the `site.p2` action again by specifying this properties file in the corresponding dialog's text field (see the preceding screenshot). This time, the p2 repository will be generated into `build/hello/org.example.site_1.0.0-eclipse.feature/site.p2` in your home directory.

You can try the p2 repository that has just been created. From Eclipse go to **Help | Install New Software...**, and in the **Work with**: text field, paste the absolute path of the created p2 repository; when you press *Enter*, you should see your **Hello SDK Feature** ready to install (see the following screenshot).
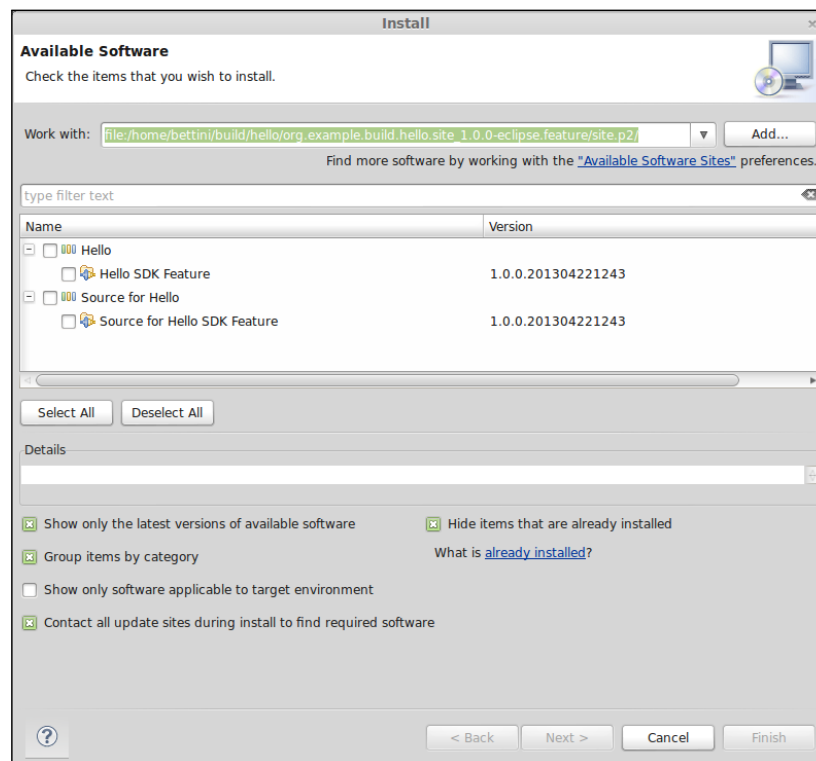
A nice thing of Buckminster is that, by default, it also generates a source feature with all the sources of all the plug-ins of your DSL implementation. If a user also installs the source feature, he/she will be able to inspect the sources of your implementation from Eclipse (without having to access its source repository to find the exact version being used).

The p2 repository you have just created can now be deployed on a remote web server; you can then publish the URL of the deployed site so that other users can install your DSL implementation like any other Eclipse feature.

You may also want to run the action `site.p2.zip`, which creates a zipped version of the p2 repository. You can put this zipped repository on the web as well to offer an offline installable version of your software (the Eclipse **Install New Software...** dialog allows you to also specify a local ZIP file as the repository).

Customizations

The projects created by the Xtext Buckminster wizard contain sensible defaults to build a p2 repository.

You can easily customize the configuration files in the projects created by the wizard. The description of the feature for your DSL implementation can be found in the `feature.xml` file in the `org.example.build.hello.sdk` feature project. Here you can change the name of the feature, the vendor, copyright information, and other things. Eclipse has a form-based editor for features, and we refer to the Eclipse documentation for the structure of this file.

A repository can potentially contain hundreds of different features to install. Eclipse p2 therefore has the concept of categories, which is used to structure features in a way that is meaningful to a user. The names of such categories are specified in the `category.xml` file of the `org.example.build.hello.site` project. You can change or customize the default categories generated by the Xtext Buckminster wizard. Eclipse provides a specific editor for `category.xml` or you can simply edit the XML file directly.

For example (see the following screenshot), we can add a description to the **main** category; this will appear in the details text area of the Eclipse **Install New Software...** dialog. You can add a corresponding description in the `main.source` category.
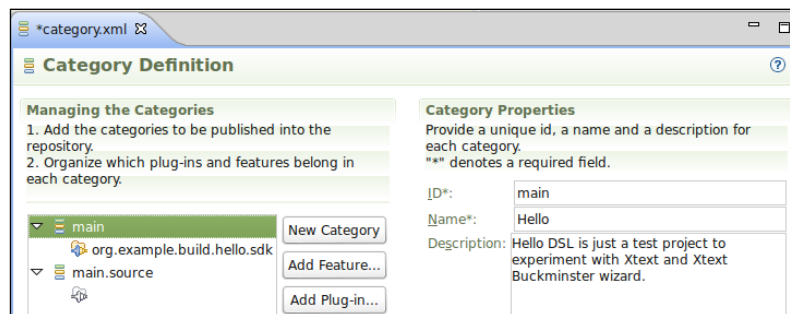
Note that the `main.source` category refers to a feature whose name is not shown, and its icon is gray. If you open `category.xml` with a text editor, you can see that it refers to the feature `org.example.build.hello.sdk.source` that does not exist. This will be created by Buckminster when running the `site.p2` action, so you can ignore it.

Defining the target platform

Although it is not strictly required to build the p2 repository in the IDE, as we said in the introduction of this chapter, a defined target platform should be used during the development in order to make sure that all the team members use the same set of required plug-ins, and also to have a better control on the requirements of the software we are developing.

Buckminster can handle standard Eclipse target definition files, and is also able to specify, resolve, and install a target platform by resolving and materializing a component through a **component query** (**CQUERY**). Without getting into technical details, the idea is that, when a component is resolved, all its dependencies are transitively resolved. The `org.example.build.hello.buckminster` project created by the Xtext Buckminster wizard, specifies in its **component specification** (**CSPEC**), two features as dependencies: `org.eclipse.xtext.sdk` and `org.eclipse.platform`. Thus, by asking Buckminster to resolve this component, a target platform which consists of these two features and all of their dependencies will be materialized. At the end of this process, the so obtained target platform will be set as the current target platform of the workspace, thus, from then on, you will be compiling and testing your projects by using only the software in this target platform. This is also the target platform that will be used in the headless build (see the next section). Unless you added additional external dependencies in your DSL implementation, these preceding two features are all you need to implement a DSL in Xtext.

To materialize the target platform, you need to open the `projects-platform.cquery` file in the `org.example.build.hello.buckminster` project with the associated Buckminster editor. You then press the button **Resolve and Materialize** and the target platform resolution will start (as illustrated in the following screenshot).
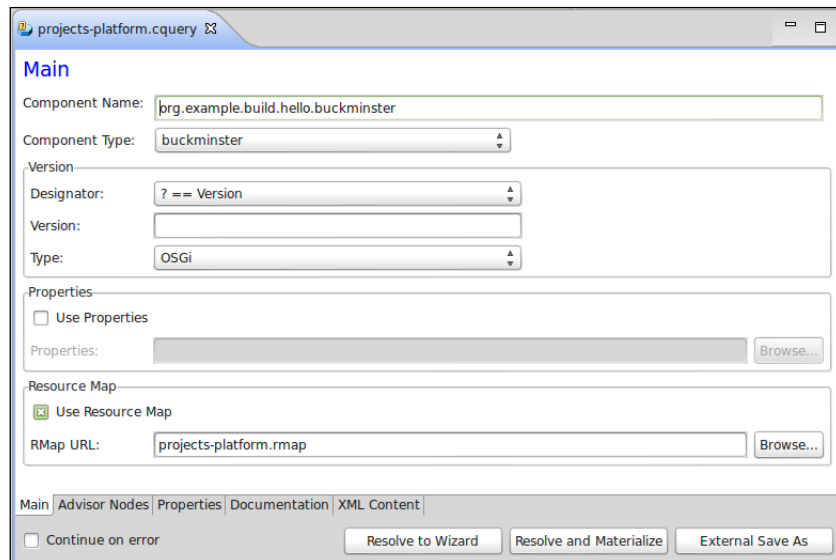
This requires an Internet connection and it might take several minutes depending on your network connection, since all the required features will be downloaded from remote p2 repositories.

Build headlessly

The Xtext Buckminster wizard also creates a `build.ant` file in the `org.example.build.hello.buckminster` project that can be used to build the p2 repository headlessly, outside Eclipse. The wizard hardcoded the path to Buckminster headless installation in the ANT file (the path you specified in the dialog of the wizard). If you plan to build your projects on a continuous integration server machine, it might be good to use a path relative to an environment variable. For example, you could modify the `buckminster.home` property in the ANT file as follows:

```
<property name="buckminster.home"
          location="${user.home}/buckminster" />
```

Remember that you can also override properties specified in an ANT file by passing a new value on the command line with the Java properties specification syntax: `D<key>=<value>`.

This ANT script will execute the headless version of Buckminster running all the commands specified in the file `commands.txt` in the `org.example.build.hello.buckminster` project.

Even without knowing Buckminster, looking at the `commands.txt` file, it should be quite easy to see what happens during the build:

1. Resolve the target platform.
2. Import all the projects in the (headless) workspace.
3. Build (that is, compile) all the projects.
4. Run the Junit tests.
5. Create the p2 repository.

Basically, the Buckminster headless process just mimics what a programmer would do in the IDE.

You can see that for running Junit tests headlessly, you just need to provide a standard Eclipse Junit launch configuration. The created `commands.txt` file refers to the launch configuration that Xtext created in your tests plug-in project. We have also used the created launch configuration for all the Junit tests of the examples shown in this book. Recall that this launch will run all the tests which are found in the source folder of the tests plug-in project. In this book we also showed how to write and run plug-in Junit tests, that require a running Eclipse. If you create a launch configuration for your plug-in Junit tests, you can add the corresponding command in the `commands.txt` file so that Buckminster will run those tests as well during the headless build (you can find such launch configurations in the sources of the examples of this book).

Note that the ANT script builds the projects and the p2 repository in a separate directory, `buildroot`, created in the same path of your projects. If you maintain your projects in a SCM repository (for example, Git, Svn, CVS), make sure to configure the repository to ignore this `buildroot` directory.

The headless build process will materialize the target platform in the directory `buildroot`. The creation of such target platform requires an Internet connection, and the first time it is created it will take some time to populate it (it will need to download Xtext SDK and the main Eclipse platform features). This means that this headless build will be independent from the Eclipse installation you used to develop your DSL; as we said at the beginning of this chapter, this is the expected behavior in a headless build. This ANT script will resolve the target platform only the first time it is run. Upon further executions, the target platform will not be recreated, thus the process takes more time only on the first execution. However, further executions will still execute a complete clean build; the headless workspace will be recreated and all the projects will be re-imported from scratch. This reflects the nature of a headless build which is usually intended as a clean build, as opposed to what happens when developing in the IDE, where the build is incremental.

Thus, with this ANT script, the building and testing of your DSL implementation can be automated, even on a continuous integration server. One of the most common open source continuous integration servers is Jenkins (`http://jenkins-ci.org/`). There is also a Buckminster plug-in for Jenkins (`https://wiki.jenkins-ci.org/display/JENKINS/Buckminster+PlugIn`). This subject is outside the scope of the book; we refer the interested reader to the Jenkins documentation for its installation and use. Once you get familiar with Jenkins, setting up a build job for your Xtext DSL project is really straightforward by relying on the projects and files generated by the Xtext Buckminster wizard.

You can use build jobs in Jenkins for your DSL to continuously test your DSL projects when new modifications are committed to the SCM repository as we do for the examples of this book, as briefly described in the next section. You can have nightly jobs that create the p2 repository and make it available on the web; the nightly builds are common to many Eclipse projects, like Xtext itself.

Further details about building Xtext projects with Buckminster, also on a continuous integration server, can be found in this blog post: `http://www.lorenzobettini.it/2012/11/building-xtext-projects-with-buckminster/`. This will give you further details about the Buckminster configuration files (namely, `.cspec`, `.cspex`, `.cquery`, and `.rmap`) which are essential to make Buckminster run a headless build. It will provide hints on how to customize such files. Of course, you can refer to the Buckminster documentation (Lindberg and Hallgren, 2010) for further and advanced details about Buckminster.

Maintaining the examples of this book

All the example DSLs shown in this book are built using Buckminster. Since we have many DSLs to build and we want a single headless build procedure to build them all, we modified the infrastructure of the projects and configuration files created by the Xtext Buckminster wizard. In particular, we have a single feature project, `org.example.site`, for building the p2 repository of all the example DSLs. Similarly, we have a single releng project, `org.example.releng`, to perform the headless build of the p2 site for the examples and to run all the Junit tests of all the DSLs.

You may want to look at all the Buckminster files in the releng project to see some variations of what we saw in this chapter. Moreover, the original `build.ant` file has been enhanced with some utility tasks like the one that automatically installs Buckminster headless if it is not present in the system where the ANT script is executed.

Furthermore, all the examples in this book are built on a Jenkins continuous integration server. The examples are maintained in a Git repository, and as soon as a commit is pushed to the remote Git repository, the corresponding Jenkins job is triggered, so that all the examples are built from scratch and all the tests are run by the Jenkins job.

Summary

Xtext also helps the programmer in the context of release engineering. Using the Xtext Buckminster wizard it is easy to build a p2 repository for releasing your DSL implementation. It is also easy to set up a headless build process that can be executed in a continuous integration server.

In the next chapter we will briefly present Xbase, a reusable expression language completely interoperable with the Java type system. When you use Xbase in your DSL you will not only inherit the grammar of its expressions, but also its Java type system, its code generator, and all its IDE aspects.

12
Xbase

In this final chapter we briefly present Xbase, a reusable expression language completely interoperable with the Java type system. By using Xbase in your DSL, you will inherit all the Xbase mechanisms for performing type checking according to the Java type system and the automatic Java code generation. Xbase also comes with many default implementations of UI aspects. The Xbase expression language is rich and has all the features of a Java-like language, such as object instantiation, method invocation, exceptions, and so on, and more advanced features such as lambda expressions and type inference. The Xtend programming language itself is built on Xbase.

Getting introduced with Xbase

As we saw throughout the examples of this book, it is straightforward to implement a DSL with Xtext. This is particularly true when you only need to care about structural aspects; in the Entities DSL of *Chapter 2*, *Creating Your First Xtext Language,* we only need to define the structure of entities. Things become more complicated when it comes to implementing expressions in a DSL; as we saw in *Chapter 8*, *An Expression Language,* we need to define many rules in the grammar using left factoring to avoid left recursion, even when we only deal with a limited number of expressions. Besides the grammar, the validation part also becomes more involved due to type checking. When we mix structures and expressions, like in the SmallJava DSL, the complexity increases again; besides advanced type checking (*Chapter 9*, *Type Checking*) we also need to take care of scoping (*Chapter 10*, *Scoping*) that is implied by relations such as inheritance. An inheritance relation also requires type conformance (subtyping) in the type system, and this introduces additional complexity.

To simplify many of these tasks, when the DSL needs to implement behavioral aspects (mainly expressions and functions or methods), Xtext provides Xbase, an expression language that can be reused in a DSL (Efftinge et al. 2012). Xbase expressions have a rich Java-like syntax, which includes standard expressions (arithmetic and Boolean), control structures (for example, if statements and loops), exceptions and object-oriented expressions (for example, method invocation and field selection). Moreover, it provides advanced features such as lambda expressions. Indeed, Xtend method bodies are based on Xbase, thus, if you use Xbase in your DSL, you will have the expressive power of Xtend expressions (however, Xtend extensions such as multi-line templates are not available in Xbase).

A DSL based on Xbase will inherit not only the syntax of such Java-like expressions, but also all its language infrastructure components, like its type checking implementation and the compiler that generates Java code. The Xbase type system is completely interoperable with the Java type system, thus, a DSL based on Xbase will be compatible with Java and its type system, including generics; your DSL will be able to seamlessly access all the Java types, which implies that you will be able to access any Java library (just like in Xtend).

> Note that Xbase only deals with expressions; your DSL will have to deal with structural features like functions and method bodies or class hierarchies.

In order to reuse Xbase Java type system in your DSL, you will just have to map the concepts of your DSL (for example, entities, attributes, and so on) into the Java model elements of Xbase (for example, classes, fields, and so on). This mapping is performed by implementing an `IJvmModelInferrer` interface. The Xbase expressions used in your DSL will then have to be associated to a Java model method, which becomes the expression's logical container. Such mapping and association will let Xbase automatically implement a proper scope for the expressions so that scoping and type checking will work out of the box. With such information, Xbase will also be able to automatically generate Java code for your DSL.

In this chapter we only give a brief introduction to Xbase (Xbase also provides an interpreter for Xbase expressions, but we will not describe it in this chapter). We refer to the Xtext official documentation, to Efftinge et al. 2012, and to the 7 languages examples, `http://www.eclipse.org/Xtext/7languages.html`. Furthermore, by navigating to **New** | **Example...** | **Xtext Examples** | **Xtext Domain-Model Example**, you can import into your workspace the Domain-Model example that ships with Xtext; this is another example of how Xbase can be used in a DSL.

> Keep in mind that when using Xbase, your DSL will be tightly coupled with Java, which might not always be what you need. We will get back to this in the final section, *Summary*.

The Expressions DSL with Xbase

For the first example of the use of Xbase, we implement a DSL similar to the Expressions DSL that we presented in *Chapter 8*, *An Expression Language*, which we call as Xbase Expressions DSL; this DSL is inspired by the **Scripting Language** DSL of the 7 languages examples.

Creating the project

First of all, we create the project:

1. Navigate to **File** | **New** | **Project...**, in the dialog navigate to the **Xtext** category and select **Xtext Project**.

2. In the next dialog you should specify the following names:

   ° **Project name**: `org.example.xbase.expressions`

   ° **Name**: `org.example.xbase.expressions.Expressions`

   ° **Extensions**: `xexpressions`

   ° Uncheck the option **Create SDK feature project**

The wizard will create three projects and it will open the grammar file `Expressions.xtext`.

Before running the MWE2 generator for the first time, you should modify the grammar so that it uses the `Xbase` grammar, not the `Terminals` grammar as we did in all previous chapters' examples:

```
grammar org.example.xbase.expressions.Expressions with
    org.eclipse.xtext.xbase.Xbase
```

Since our grammar now inherits from the `Xbase` grammar, all the `Xbase` grammar rules are in effect in our DSL.

> When using Xbase, the generated Java files in the `src-gen` folder will contain lots of warnings of the shape:
>
> ```
> Discouraged access: The method ... from the type ...
> is not accessible due to restriction on required
> library org.eclipse.xtext.xbase_....jar
> ```
>
> This is due to the fact that Xbase API is still provisional and it will be subject to changes in the future possibly breaking backward compatibility. Of course you can ignore these warnings. When a new version of Xbase is released you might have to run the MWE2 workflow again, and possibly modify your code to be compatible with the new version. In any case, a test suite is also required to make sure that your code is still working with the new version of Xbase.

We want to recreate a DSL similar to the Expressions DSL we introduced in *Chapter 8*, *An Expression Language*. Thus, we want to be able to both declare variables and to write expressions like additions, multiplications, and so on. Thus, a program in this DSL is like a big code block. Xbase has a specific rule for dealing with code blocks: `XBlockExpression`.

> The most generic form of Xbase expression, which also corresponds to the base class of all Xbase expressions, is `XExpression`.

Since Xbase has its own validator that implements lots of useful constraint checks, and since, like all the validators we implemented in this book, it is based on the types of the elements being validated, we should make the main rule for our DSL return an `XBlockExpression` object. This way, our DSL root element will be conformant to `XBlockExpression` and it will be automatically validated accordingly by Xbase. In the Expressions DSL of *Chapter 8*, *An Expression Language,* we also made sure that variable references are validated so that a variable can only refer to variables already defined; this check is not necessary anymore when we use Xbase since it is automatically implemented for the `XBlockExpression` elements.

Thus, we start writing the grammar as follows:

```
grammar org.example.xbase.expressions.Expressions with
  org.eclipse.xtext.xbase.Xbase

generate expressions "http://www.example.org/xbase/expressions/
Expressions"

import "http://www.eclipse.org/xtext/xbase/Xbase"

ExpressionsModel returns XBlockExpression:...
```

Remember that when we use `returns` in a rule, we specify a type, not a rule name. Since this type, `XBlockExpression`, is defined in Xbase EMF metamodel, we need to import this metamodel. We do this by specifying the EMF namespace corresponding to Xbase (recall that each Xtext grammar defines an EMF namespace; it is the one defined after the `generate` section).

Now, we want our `ExpressionsModel` to consist of variable declarations and single expressions. Xbase has a specific rule for that: `XExpressionInsideBlock`; thus, we can simply reuse that rule:

```
ExpressionsModel returns XBlockExpression:
  {ExpressionsModel}
    (expressions+=XExpressionInsideBlock)*;
```

With this simple grammar, we have a working parser that allows us to write programs consisting of variable declarations and single expressions. These have the same syntax as Xtend expressions. If you want to experiment with this DSL, you can start Eclipse, create a plug in project in the workspace, and, in the `src` folder, create a new `.xexpressions` file (remember to accept the option to add the Xtext nature to the project). You should also add the bundle `org.eclipse.xtext.xbase.lib` as a dependency in the *Dependencies* section of the `MANIFEST.MF` editor.

The IJvmModelInferrer interface

Besides the grammar rules which can be reused in a DSL, the interesting feature of Xbase is the integration with Java. The same integration you have already experienced in Xtend can be reused in any DSL based on Xbase. This means that the entire type system of Xbase, which corresponds to the type system of Java, together with the additional type inference mechanisms that you enjoyed in Xtend, can be a part of your DSL as well.

However, in order to reuse the Xbase type system in your DSL, it is not enough to use Xbase grammar rules; you also need to give your model elements a "context" so that Xbase can check your model elements according to that context. You basically need to tell Xbase how your model elements map to Java elements. The Xbase expressions which are contained in your model elements will then be typed and checked by Xbase as parts of the corresponding Java elements.

You basically have to map your model elements to Java types (classes and interfaces), fields and methods. This mapping is a model to model mapping, since you map your DSL model elements to a model that represents Java elements. When you map a model element to a Java method, you can specify that its body is represented by an Xbase expression. Given that, Xbase will be able to type and check that Xbase expression, since it will consider it in the context of a Java method (which in turn is part of a Java class, which can extend another Java class, and so on.). If that Xbase expression uses `this`, Xbase will know what it refers to; similarly for `super`, method parameters, and so on. Unless you need to do some custom things with Xbase expressions, just specifying this mapping to Java elements will be enough; you will not need to provide a custom scoping, since Xbase will be able to compute the scope using the mapped Java elements.

This mapping is specified by implementing an `IJvmModelInferrer` interface; since we use Xbase in our grammar, the MWE2 workflow generates an Xtend stub class, `ExpressionsJvmModelInferrer` in the `jvmmodel` subpackage. The stub class has an empty dispatch method named `infer`, which is the method where you specify the mapping to Java elements:

```
def dispatch void infer(ExpressionsModel model,
    IJvmDeclaredTypeAcceptor acceptor,
    boolean isPreIndexingPhase)
```

In this method you will create Java model elements, associate them to your DSL elements, and pass them to the `acceptor`; this implements the mapping. The Java elements themselves can be created using the injected extension `JvmTypesBuilder`. This provides a useful API to create Java model elements, for instance, `toClass`, `toMethod`, `toField`, and so on. All these methods take the source element as the first parameter. Besides creating a Java model element, these methods also record the association with the original source element. Xtext uses this association to provide a default implementation of many UI concepts.
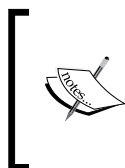
> Note that with `JvmTypesBuilder` you will not create effective Java elements (for example, `java.lang.Class`, `java.lang.reflect.Method`, `java.lang.reflect.Field`, and so on), but their representation in the Xbase EMF model for Java elements (for example, `JvmGenericType`, `JvmOperation`, `JvmField`, and so on).

For our DSL we can specify the mapping as: given an `ExpressionsModel` object, we map it to a Java class with a single `main` method; any Xbase expressions in our `ExpressionsModel` will be associated to the body of the `main` method. In particular, since `ExpressionsModel` is itself an `XBlockExpression` object, we can directly associate the `ExpressionsModel` object itself with the body of the `main` method. The implementation of our inferrer is as follows:

```
class ExpressionsJvmModelInferrer extends AbstractModelInferrer {
  @Inject extension JvmTypesBuilder

  def dispatch void infer(ExpressionsModel model,
              IJvmDeclaredTypeAcceptor acceptor,
              boolean isPreIndexingPhase) {
    val className =
      model.eResource.URI.trimFileExtension.lastSegment
    acceptor.accept(model.toClass(className)).
    initializeLater [
      members += model.toMethod('main',
                          model.newTypeRef(Void::TYPE)) [
        parameters += model.toParameter
          ("args", model.newTypeRef(typeof(String)).
                addArrayTypeDimension)
        static = true
        // Associate the model with the body of the main method
        body = model
      ]
    ]
  }
}
```

Let us comment on this code. First of all, we name the Java class after the input file (without the file extension). Note that the class is created using the `JvmTypesBuilder` method `toClass`. However, further initialization operations concerning this class are deferred (`initializeLater`). In fact, the inferrer is used by Xtext first on the indexing phase (see *Chapter 10*, *Scoping*) and again after the indexing phase has finished. The method `initializeLater` is called in the second step. In the first step, the index has not been completely built, thus you cannot rely on cross-references having been resolved. Thus in the indexing phase you just create Java classes or interfaces for your model elements; after the indexing has finished, you can add elements to the created Java types.

In the previous code, we add to the Java class a method using `JvmTypesBuilder.toMethod`, passing the name of the method and its return type. When we specify types for Java model elements we must always use instances of class `JvmTypeReference`. To create a type reference to an existing Java type we use the method `JvmTypesBuilder.newTypeRef` that takes a Java `Class` object; for the `main` method, the return type must be void, thus we use the static instance `java.lang.Void.TYPE`.

Once we created a Java method, we can further initialize it with a lambda that gets the created method as parameter (formally, an instance of `JvmOperation`). In this example, we add to the created Java method a parameter named `args` of type `String[]` (note that the use of `JvmTypesBuilder` API); we also specify that the method is `static`.

Finally, we associate the whole Xbase expression of the program with the body of the method (recall that `ExpressionsModel` is an `XBlockExpression`). As said before, this will allow Xbase to build a proper scope for the expressions and to perform type checking and validation of expressions.

You can now restart Eclipse and try the editor for this DSL again (we assume you have already created the project as detailed before). The result is shown in the following screenshot. Here you can see that in the `xexpressions` file we can declare variables with the Xbase syntax, which corresponds to the same syntax for variable declaration in Xtend:
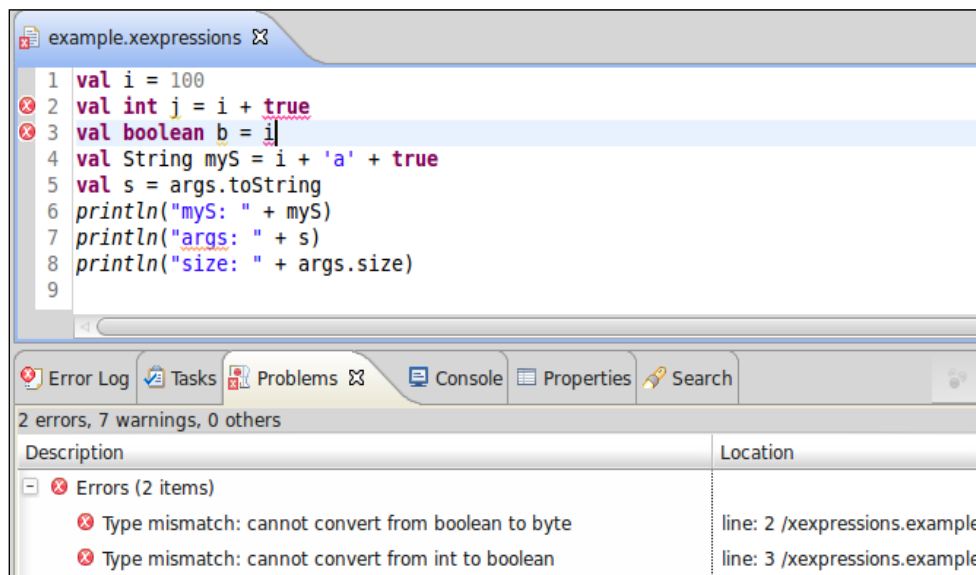
Note that all the type checking is automatically performed by Xbase. In the preceding screenshot you can see two errors due to wrong types in the expressions. Note also that the conversion to string type, when using the operator +, is performed automatically. Indeed, besides the `IJvmModelInferrer` interface, we did not have to specify anything else, neither a custom scoping nor a custom validator.

Moreover, although the `args` variable is not defined anywhere in our program, we can still refer to it since the whole expression block has been associated with the body of a method with the parameter `args`. The association we specified in the inferrer corresponds to the fact that the parameter `args` is available in the program with the specified type. Thus, since in the inferrer we specified that `args` is an array of `String`, we can call the method `size` on `args` in our program as follows:

```
println("size: " + args.size)
```

Code generation

When using Xbase we get code generation for free. This can be observed by saving an error-free `.xexpressions` file in the IDE. When you do so, a `src-gen` folder is automatically created in your project (you may want to make this new folder a source folder of your Java project: right-click on the folder, then navigate to **Build Path | Use as Source Folder**). We have already seen this behavior in *Chapter 5*, *Code Generation*, when we wrote a code generator for the Entities DSL. However, this time, we did not write any code generator.

This automatic Xbase code generator mechanism is based on the `IJvmModelInferrer` interface; the code generator uses the mapped Java model to generate Java code. See the following screenshot:

This is an extremely useful feature, since when using Xbase we only need to write the `IJvmModelInferrer` interface correctly. Xbase will take care of all the rest, including type checking and code generation.

> The automatic code generation mechanism is performed by `JvmModelGenerator`, which is automatically bound in the Guice module generated during the MWE2 workflow. Of course, you could still write your own code generator and override the Guice binding; it is recommended to rely on the automatic Xbase code generator though since writing a code generator manually for a DSL that uses Xbase would require some effort.

Debugging

With Xbase you can debug your DSL sources instead of the generated Java code—just like when working with Xtend. This is another valuable feature you get for free.

You can try and set a break pointer in the `.xexpressions` program and then right-click on the generated Java file and navigate to **Debug As | Java Application** (this menu is available since the generated Java file contains a `main` method). Now, when the execution reaches the line corresponding to the set breakpoint in the `.xexpressions` file, you will see that the debugger perspective is opened with the Expressions editor and the "Variables" view shows the contents of the variables of your program (see the following screenshot). Just like when debugging Xtend code, if you need to debug the generated Java code, you can do so by right-clicking on the **Debug** view and by navigating to **Show Source | Java**:
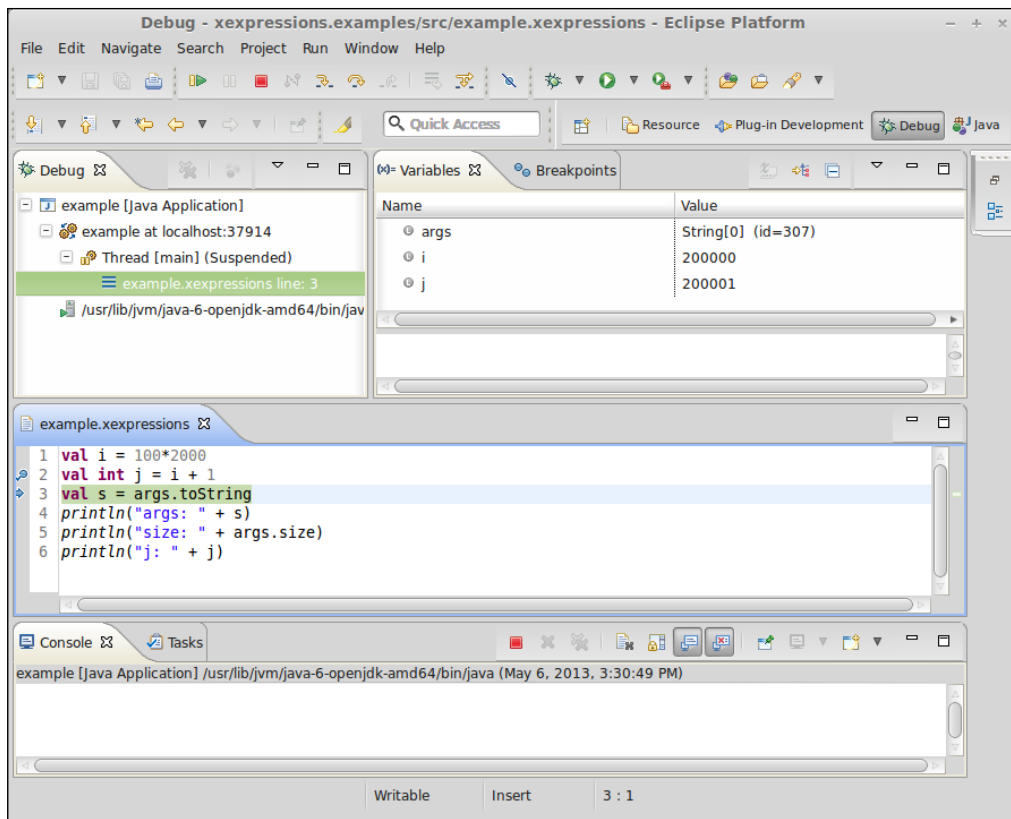
The Entities DSL with Xbase

We will now implement a modified version of the Entities DSL we implemented in *Chapter 2*, *Creating Your First Xtext Language* using Xbase. This will allow us to implement a more involved DSL where, inside entities, we can also write operations apart from attributes (this is inspired by the Xtext Domain-Model example).

Creating the project

As usual we create the Xtext project by performing the following steps:

1. Navigate to **File | New | Project...**, in the dialog go to the **Xtext** category and select **Xtext Project**.

2. In the next dialog you should specify the following names:
   - **Project name**: `org.example.xbase.entities`
   - **Name**: `org.example.xbase.entities.Entities`
   - **Extensions**: `xentities`
   - Uncheck the option **Create SDK feature project**

The wizard will create three projects and it will open the grammar file `Entities.xtext`.

Before running the MWE2 generator for the first time, you should modify the grammar so that it uses the `Xbase` grammar:

```
grammar org.example.xbase.entities.Entities with
  org.eclipse.xtext.xbase.Xbase
```

Defining attributes

We define the rules for attributes by using some rules inherited from the Xbase grammar:

```
grammar org.example.xbase.entities.Entities with org.eclipse.xtext.
xbase.Xbase

generate entities "http://www.example.org/xbase/entities/Entities"

Model:
  entities+=Entity*;

Entity:
'entity' name=ID ('extends' superType=JvmParameterizedTypeReference)?
'{'
  attributes += Attribute*
'}';

Attribute:
  'attr' (type=JvmTypeReference)? name=ID
  ('=' initexpression=XExpression)? ';';
```

The rule for `Entity` is similar to the corresponding rule of the Entities DSL of *Chapter 2, Creating Your First Xtext Language*. However, instead of referring to another `Entity` in the feature `superType`, we refer directly to a Java type; since Xbase implements the Java type system, an entity can extend any other Java type. Moreover, since an `Entity` will correspond to a Java class (we will implement this mapping in the inferrer), it will still be able to have an entity as a super type, though it will specify it through the corresponding inferred Java class.

We refer to a Java type using the Xbase rule `JvmParameterizedTypeReference`. As the name of the rule suggests, we can also specify type parameters, for instance, we can write:

```
entity MyList extends java.util.LinkedList<Iterable<String>> {}
```

Similarly, for attributes, we use Java types for specifying the type of the attribute. In this case we use the Xbase rule `JvmTypeReference`; differently from `JvmParameterizedTypeReference`, this rule also allows to specify types for lambdas. Thus, for instance, we can define an attribute as follows:

```
attr (String,int)=>Boolean c;
```

We also allow to specify an initialization expression for an attribute using a generic Xbase expression, `XExpression`. Note that both the type and the initialization expression are optional; this design choice will be clear after looking at the model inferrer:

```
class EntitiesJvmModelInferrer extends AbstractModelInferrer {
  @Inject extension JvmTypesBuilder
  @Inject extension IqualifiedNameProvider

  def dispatch void infer(Entity entity,
        IJvmDeclaredTypeAcceptor acceptor, boolean isPreIndexingPhase)
{
    acceptor.accept(entity.toClass("entities."+entity.name)).
    initializeLater [
      documentation = entity.documentation
      if (entity.superType != null)
        superTypes += entity.superType.cloneWithProxies
      entity.attributes.forEach[
          a |
          val type = a.type ?: a.initexpression?.inferredType
          members += a.toField(a.name, type) [
              documentation = a.documentation
              if (a.initexpression != null)
                initializer = a.initexpression
          ]
```

```
          members += a.toGetter(a.name, type)
          members += a.toSetter(a.name, type)
      ]
    ]
  }
}
```

Note that in this example we provide an `infer` method for `Entity`, not
for the root `Model`. In fact, the default implementation of the superclass
`AbstractModelInferrer` can be summarized as follows:

```
public void infer(EObject e, ...) {
  for (EObject child : e.eContents()) {
    infer(child, acceptor, preIndexingPhase);
  }
}
```

It simply calls the method `infer` on each element contained in the root of the
model. Thus, we only need to provide a dispatch method `infer` for each type of
our model that we want to map to a Java model element. In the previous example
we needed to map the whole program (that is, the root model element), while in
this example we map every entity of a program.

As we did in the previous section, we use an inject `JvmTypesBuilder` extension
to create the Java model elements and associate them with the elements of our DSL
program AST.

First of all, we specify that the the superclass of the mapped class will be the entity's
`superType` if one is given.
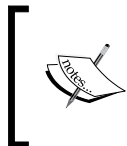
> Note that we clone the type reference of `superType`. This is required
> since `superType` is an EMF containment reference. The referred
> element can be contained only in one container, thus, without the clone,
> the feature `superType` would be set to null after the assignment.

For each attribute, we create a Java field using `toField` and a getter and
setter method using `toGetter` and `toSetter` respectively (these are part of
`JvmTypesBuilder`). If an initialization expression is specified for the attribute,
the corresponding Java field will be initialized with the Java code corresponding
to the `XExpression`.

The interesting thing in the mapping for attributes is that we use Xbase features for type inference; if no type is specified for the attribute, the type of the Java field will be automatically inferred by Xbase using the type of the initialization expression. If neither the type nor the initialization expression is specified, Xbase will automatically issue an error. Of course, if we specify both the type of the attribute and its initialization expression, Xbase will automatically check that the type of the initialization expression is conformant to the declared type. The following screenshot shows two validation errors issued by Xbase:
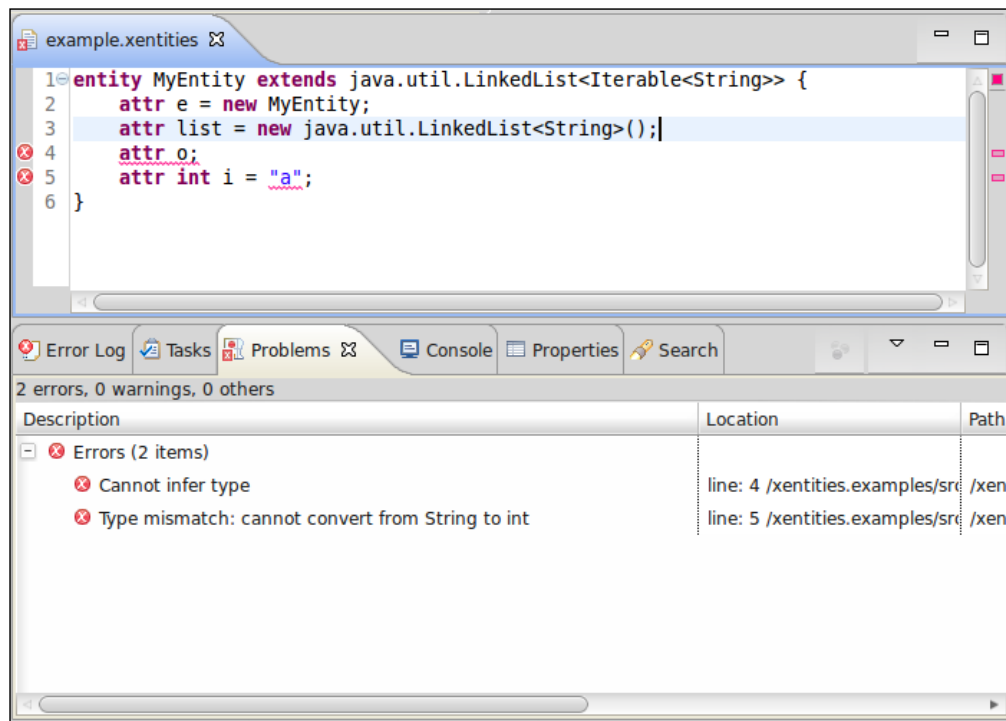
Both for the mapped Java model class and Java model field we set the `documentation` feature using the documentation attached to the program element. This way, if in the program we write a comment with /* */ before an entity or an attribute, in the generated Java code this will correspond to a `Javadoc` comment, as illustrated in the following screenshot:

```
example.xentities ⊠
1  /* my entity */
2⊖ entity MyEntity {
3      /* my attribute */
4      attr String s;|
5  }
```

```
J MyEntity.java ⊠
1⊖ /**
2   * my entity
3   */
4  @SuppressWarnings("all")
5  public class MyEntity {
6⊖   /**
7     * my attribute
8     */
9    private String s;
10
11⊖  public String getS() {
12     return this.s;
13   }
14
15⊖  public void setS(final String s) {
16     this.s = s;
17   }
18 }
```
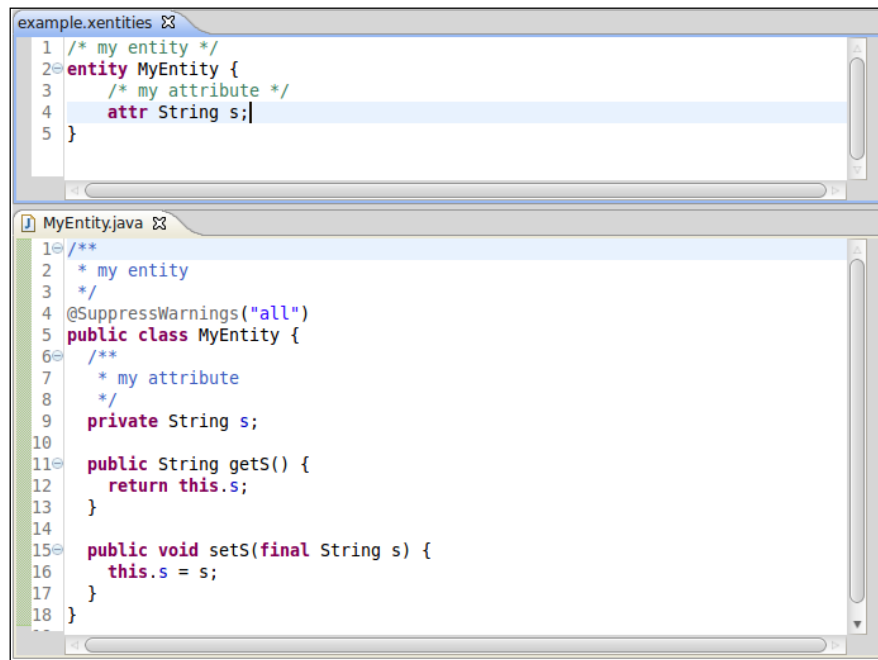
Defining operations

Now we add operations to our entities, which will correspond to Java methods. Thus, we add the rule for `Operation` (to keep the example simple and to concentrate on Xbase, we did not introduce an abstract element for both attributes and operations, and we require that operations are specified after the attributes):

```
Entity:
'entity' name=ID ('extends' superType=JvmParameterizedTypeReference)?
'{'
    attributes += Attribute*
    operations += Operation*
'}';

Operation:
'op' (type=JvmTypeReference)? name=ID
'(' (params+=FullJvmFormalParameter (','
```

```
    params+=FullJvmFormalParameter)*)? ')'
  body=XBlockExpression;
```

Here we use the Xbase rule `FullJvmFormalParameter` to specify parameters; parameters will have the syntactic form of Java parameters, that is, a `JvmTypeReference` stored in the feature `parameterType` and a `name`. The body of an operation is specified using the Xbase rule `XBlockExpression`, which also requires the curly brackets.

In our inferrer we add the mapping to a Java model method with the following code:

```
entity.operations.forEach[
  op |
  members += op.toMethod(op.name, op.type ?: inferredType) [
    documentation = op.documentation
    for (p : op.params) {
      parameters += p.toParameter(p.name, p.parameterType)
    }
    body = op.body
  ]
]
```

This is similar to what we did in the first example of this chapter; however, this time, we have a corresponding element in our DSL, `Operation`. Thus, we create a Java model parameter for each parameter defined in the program, and we use the `Operation` instance's body as the body of the mapped Java model method. Also for the operation the return type can be omitted: in that case, the corresponding Java model method will have the return type that Xbase infers from the operation's `XBlockExpression`.

The association of the method's body with the operation's body implicitly defines the scope of the `XBlockExpression` object. Since an operation is mapped to a non-static Java method, in the operation's expressions you can automatically refer to the attributes and operations of the containing entity and of the entity's `supertype` (since they are mapped to Java fields and methods, respectively).

This can be seen in the following screenshot, where the operation accesses the entity's fields and the inherited method `add`; Xbase automatically adds other tooling features, such as information hovering and the ability to jump to the corresponding Java method:
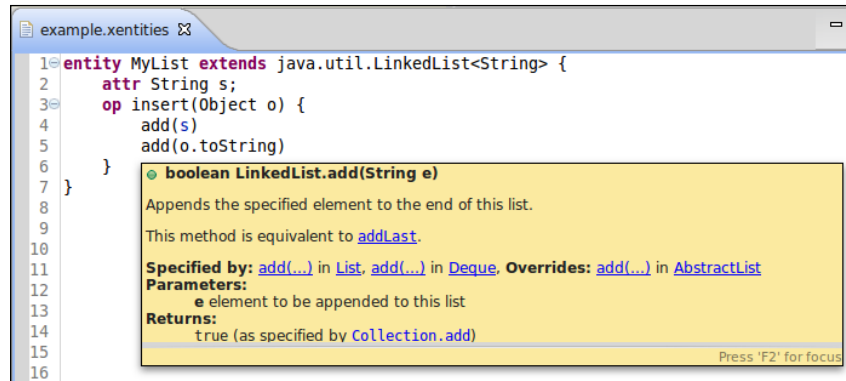
In fact, the scope for `this` is implied by the association between the operation and the Java model method. The same holds for `super`, as shown in the following example, where the operation `m` overrides the one in the `supertype` and can access the original version with `super`:

```
entity Base {
  op m() { "Base" }
}

entity Extended extends Base {
  op m() { super.m() + "Extended" }
}
```

Indeed, associating an Xbase expression with the body of a Java model method corresponds to making the expression **logically contained** in the Java method. This logical containment defines the scope of the Xbase expression.

> An Xbase expression can have only one logical container.

In the examples we have shown so far, we have always associated an `XExpression` with the body of a Java method or with the initializer of a field. There might be cases where you want to add a method to the Java model that does not necessarily correspond to an element of the DSL.

In these cases, you can specify a lambda as the body of the created method; the lambda takes an `ITreeAppendable` object as a parameter. You can use this appendable object to specify a string which will correspond to the method body in the generated Java code. For instance, we can add to the mapped Java class a `toString` method as follows:

```
members += entity.toMethod("toString", entity.
newTypeRef(typeof(String))) [
  body = [
    append(
    '''
    return
    "entity «entity.name» {\n" +
      «FOR a : entity.attributes»
          "\t«a.name» = " + «a.name».toString() + "\n" +
      «ENDFOR»
    "}";'''
    )
  ]
]
```

For instance, given this entity definition:

```
entity C {
  attr List l;
  attr s = "test";
}
```

The generated Java method will be:

```
public String toString() {
    return
    "entity C {\n" +
          "\tl = " + l.toString() + "\n" +
          "\ts = " + s.toString() + "\n" +
    "}";
}
```

As usual, remember to write Junit tests for your DSL (see *Chapter 7*, *Testing*); for instance, this test checks the execution of the generated `toString` Java method:

```
class EntitiesCompilerTest {

  @Inject extension CompilationTestHelper
  @Inject extension ReflectExtensions

  @Test
  def void testGeneratedToStringExecution() {
```

```
      '''
    entity C {
      attr l = newArrayList(1, 2, 3);
      attr s = "test";
    }'''.compile[
      val obj = it.compiledClass.newInstance
      '''
      entity C {
        l = [1, 2, 3]
        s = test
      }'''.toString.assertEquals(obj.invoke("toString"))
    ]
}
```

> The `JvmTypesBuilder` class provides methods to automatically
> add into the inferred class methods such as `toString`, `equals`,
> and `hashCode` (`toToStringMethod`, `toEqualsMethod`,
> and `toHashCodeMethod`, respectively). You should use
> the `JvmTypesBuilder` instance's methods in your DSL if you want
> to add the corresponding methods in the generated Java class, since
> they generate a sensible implementation. In this chapter, we showed
> a manual implementation of `toString` just for demonstration.

There are many advanced features available when using Xbase that are outside
the scope of this book. Many are illustrated in the "7 languages examples" and other
examples available online, for example, `https://github.com/LorenzoBettini/`
`Xtext2-experiments`.

Imports

We saw in *Chapter 10*, *Scoping,* that Xtext provides support for namespace-based
imports for qualified names. Xbase provides an automatic mechanism for imports
of Java types and many additional tooling for the Eclipse editor. To include such
feature in a DSL that uses Xbase it is enough to use the rule `XImportSection`; for
instance, in our Xbase Entities DSL we modify the root rule as follows:

```
Model:
  importSection=XImportSection?
  entities+=Entity*;
```

The addition of this rule automatically adds to the DSL the same import
functionalities that you saw in Xtend: in addition to the standard Java import
mechanisms (including static imports), you can now write `import static`
`extension` statements for static methods and those static methods will be
available as extension methods in the program.

Besides this enhanced import statements, Xbase adds nice tooling functionalities in the editor of the DSL that reflect the ones of JDT and Xtend; a few of them are listed in the following bullet list:

- Warnings for unused imported types
- **An Organize Imports** menu (available also with the keyboard shortcut *Ctrl + Alt + O*)
- Automatic insertion of import statements (when you use the autocompletion for specifying a Java type reference in the program, the corresponding import statement is automatically added)

> Remember to merge the files `plugin.xml` and `plugin.xml_gen` in the UI project after running the MWE2 workflow; this is required to add the **Organize Imports** menu items in the editor.

Customizations

Depending on the complexity of your DSL, you might want to customize the default implementation of several components of Xbase, such as scoping and validation. This can be achieved by subclassing the corresponding Xbase classes and by specifying the bindings in the Guice module. Note that the Xbase scope provider implementation is not based on `AbstractDeclarativeScopeProvider`, thus, you cannot rely on the convention for method names that we used in *Chapter 10*, *Scoping*; you will have to redefine the `getScope` method. Moreover, you will have to inspect the implementation of Xbase scope provider and validator to understand which method to override.

It is also possible to override some rules of the Xbase grammar in order to change the syntactic shape of some expressions. Moreover, you can change an Xbase expression rule in order to add syntax for new expressions which are specific to your DSL. In this case, you also need to provide custom implementations for some classes of Xbase, such as the type provider and the compiler. In particular, you need to specify how to type your expressions and how they compile into Java. We refer to the official Xtext documentation and to the 7 languages examples for these advanced topics.

Summary

Xbase can add a powerful Java-like expression language to your DSL, and by implementing the model inferrer, you will automatically reuse the Xbase Java type system implementation and the generation of Java code. This does not imply that all the concepts described in all the previous chapters (validation, code generation, scoping, and so on) are useless. In fact, knowing the main concepts underlying Xtext is required to effectively implement a DSL even when using Xbase. Moreover, in case you need to modify or add expressions to a DSL that uses Xbase, you may have to provide a custom scoping and validation as well. When using Xbase, your DSL will be tightly coupled with Java, which might not always be what you need. Your DSL could be used only for writing specifications or simpler structures and in that case you will not need Xbase expressions; these would only add unwanted complexity. Alternatively, your DSL might not be bound to Java and might require code generation into another target language; also in this case you cannot use Xbase.

I hope that you enjoyed this book as much as I enjoyed writing it! If you had never used Xtext before, I hope that this book gave you enough knowledge to get started and get productive in developing DSLs with Xtext and Xtend. If you were already familiar with Xtext, I hope that this book increased your knowledge about this framework and that the methodologies and best practices illustrated throughout the chapters will make you more productive.

Keep your code clean and well tested!

Bibliography

- *Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D., Compilers: principles, techniques, and tools*, 2nd Edition, (2007), *Addison-Wesley.*

- *Beck, K., Test Driven Development: By Example* (2002), *Addison-Wesley.*

- *Brown, D., Levine, J., Mason, T., lex & yacc* (1995), *O'Reilly.*

- *Cardelli, L., Type Systems* (1996), *ACM Computing Surveys, 28(1):263–264.*

- *Efftinge, S., Eysholdt, M., Köhnlein. J., Zarnekow, S., Hasselbring, W., von Massow, R., Hanus, M., Xbase: Implementing Domain-Specific Languages for Java (2012), Proceedings of the 11th International Conference on Generative Programming and Component Engineering, 112-121, ACM.*

- *Fowler, M., Inversion of Control Containers and the Dependency Injection pattern* (2004), `http://www. martinfowler.com/articles/injection.html.`

- *Fowler, M., Continuous Integration* (2006), `http://martinfowler.com/articles/continuousIntegration.html.`

- *Fowler, M., Domain-Specific Languages* (2010), *Addison-Wesley.*

- *Gamma, E., Helm, R., Johnson, R.,* and *Vlissides, J., Design Patterns: Elements of Reusable Object-Oriented Software* (1995), *Addison-Wesley.*

- *Ghosh, D., DSLs in Action* (2010), *Manning.*

- *Hindley, J.R., Basic Simple Type Theory* (1987), *Cambridge University Press.*

- *Köhnlein, J., Xtext tip: How do I get the Guice Injector of my language?* (2012), `http://koehnlein.blogspot.it/2012/11/xtext-tip-how-do-i-get-guice-injector.html.`

- *Levine, J., Flex & Bison: Text Processing Tools* (2009), *O'Reilly.*

- *Lindberg, H.* and *Hallgren, T., Eclipse Buckminster; The Definitive Guide* (2010).

- *Martin, R.C., Agile Software Development, Principles, Patterns, and Practices* (2002), *Prentice Hall.*

- *Martin, R.C., Clean Code: A Handbook of Agile Software Craftsmanship (2008), Prentice Hall.*

- *Martin, R.C., The Clean Coder: A Code of Conduct for Professional Programmers (2011), Prentice Hall.*

- *Parr, T., The Definitive ANTLR Reference: Building Domain-Specific Languages (2007), Pragmatic Programmers.*

- *Pierce, B.C., Types and Programming Languages (2002), The MIT Press, Cambridge, MA.*

- *Steinberg, D., Budinsky, F., Paternostro, M.,* and *Merks, E., EMF: Eclipse Modeling Framework,* 2nd Edition, *(2008), Addison-Wesley.*

- *Vlissides, J., Generation Gap [software design pattern] (1996), C++ Report, 8(10),* p. *12, 14-18.*

- *Voelter, M., DSL Engineering: Designing, Implementing and Using Domain-Specific Languages (2013).*

- *Zarnekow, S., Xtext Best Practices (2012), EclipseCon Europe,* `http://www.eclipsecon.org/europe2012/sessions/xtext-best-practices`.

- *Zarnekow, S., Xtext Corner #8 - Libraries Are Key (2012-b),* `http://zarnekow.blogspot.it/2012/11/xtext-corner-8-libraries-are-key.html`.

Index

U

UI
  improving 225-227
UI testing
  about 138
  content assist, testing 138-140
  editor, testing 142-144
  testing frameworks 145
  workbench integration, testing 140, 142
utility methods 202, 203

V

validation rules
  about 205
  cyclic class hierarchies 205, 206
  duplicates, checking for 212, 214
  member selections, checking 207, 208
  return statements, checking 209-212
validation, Xtext
  about 71
  custom validators 74-77
  default validators 72, 73
validator
  about 179-183
  testing 125-127

W

warning method 78
well-typed 174
with operator 64
workbench integration
  testing 140-142

X

Xbase
  about 295, 296
  Entities DSL 305
  Expressions DSL 297
Xtend
  additional operators 64, 65
  benefits 50
  code generator, writing 88, 90
  extension methods 52-54
  features, expression 50
  features, type inference 50
  it variable 55
  lambda expression 55, 56
  polymorphic method invocation 66
  switch expressions 66, 68
xtend method 62
Xtend SDK 2.4.2 18
Xtext
  about 17, 229
  cross-reference resolution 229
  documentation 23
  Google Guice 102, 103
  installing 18
  validation 71
Xtext Buckminster wizard
  about 283
  building headlessly 290-292
  customizations 288, 289
  p2 repository, building 286, 287
  target platform, defining 289, 290
  using 283-285
Xtext generator 33-35
XtextProjectHelper class 140
Xtext SDK 2.4.2 18

Thank you for buying
Implementing Domain-Specific Languages
with Xtext and Xtend

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.
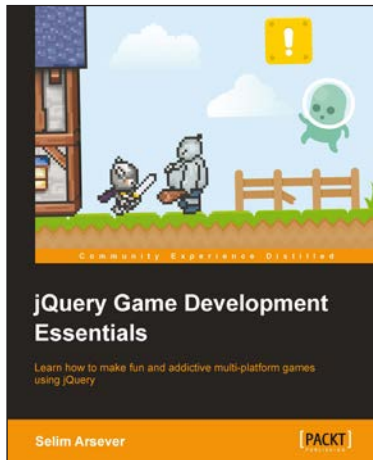
jQuery Game Development Essentials

ISBN: 978-1-84969-506-0   Paperback: 244 pages

Learn how to make fun and addictive multi-platform games using jQuery

1. Discover how you can create a fantastic RPG, arcade game, or platformer using jQuery!

2. Learn how you can integrate your game with various social networks, creating multiplayer experiences and also ensuring compatibility with mobile devices.

3. Create your very own framework, harnessing the very best design patterns and proven techniques along the way.
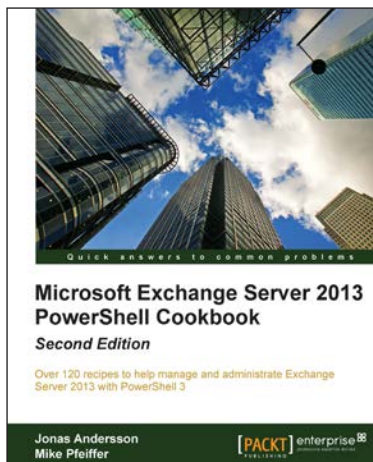
## Microsoft Exchange Server 2013 PowerShell Cookbook: Second Edition

ISBN: 978-1-84968-942-7   Paperback: 504 pages

Over 120 recipes to help manage and administrate Exchange Server 2013 with PowerShell 3

1. Newly updated and improved for Exchange Server 2013 and PowerShell 3

2. Learn how to write scripts and functions, schedule scripts to run automatically, and generate complex reports with PowerShell

3. Manage and automate every element of Exchange Server 2013 with PowerShell such as mailboxes, distribution groups, and address lists

Please check **www.PacktPub.com** for information on our titles
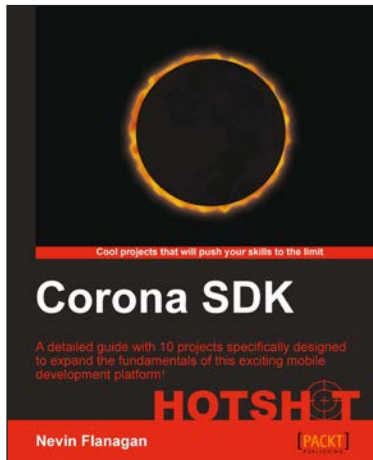
Corona SDK Hotshot

ISBN: 978-1-84969-430-8   Paperback: 334 pages

A detailed guide with 10 projects specifically designed to expand the fundamentals of this exciting mobile development platform!

1. Ten fully developed code projects that build on previous projects and present new techniques.

2. Freely reusable art and sound files included with every project help you jumpstart your own development.

3. Numerous advanced techniques to make the most out of Corona's features and the Lua programming language.
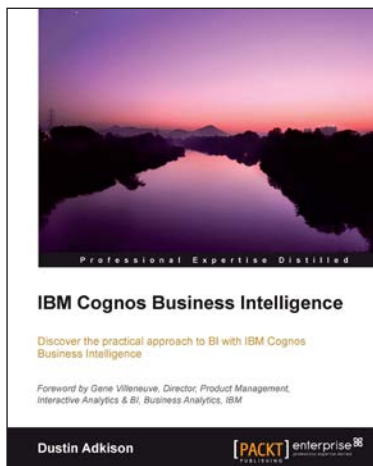
IBM Cognos Business Intelligence

ISBN: 978-1-84968-356-2   Paperback: 318 pages

Discover the practical approach to BI with IBM Cognos Business Intelligence

1. Learn how to better administer your IBM Cognos 10 environment in order to improve productivity and efficiency.

2. Empower your business with the latest Business Intelligence (BI) tools.

3. Discover advanced tools and knowledge that can greatly improve daily tasks and analysis.

4. Explore the new interfaces of IBM Cognos 10.

Please check **www.PacktPub.com** for information on our titles