A Type Graph Model for Java Programs

Arend Rensink and Eduardo Zambon

February 9, 2009

Abstract

In this report we present a type graph that models all executable constructs of the Java programming language. Such a model is useful for any graph-based technique that relies on a representation of Java programs as graphs. The model can be regarded as a common representation to which all Java syntax graphs must adhere. We also present the systematic approach that is being taken to generate syntax graphs from Java code. Since the type graph model is comprehensive, i.e., covers the whole language specification, the technique is guaranteed to generate a corresponding graph for any valid Java program. In particular, we want to extract such syntax graphs in order to perform static analysis and model checking of programs written in Java. Although we focus on Java, this same approach could be adapted for other programming languages.

Contents

1 Introduction **3**

2 Preliminaries and core concepts **4**

3 Description of approach taken **6**

3.1 Creating the type graph 7

3.2 Constructing syntax graphs from code 8

4 Type graph model **9**

4.1 Class diagram 9

4.2 Node types, attributes and associations 9

4.3 Example of a syntax graph 12

5 Related work **14**

6 Conclusion and future work **15**

A Comprehensive list of node types of the type graph **18**

Chapter 1

Introduction

A graph is a flexible structure that is used to represent several different artifacts in computer science. However, the mathematical definition of a graph alone does not allow us to restrict a representation to a certain pattern or form. Such restrictions can be enforced by means of a *type graph*, a model that describes rules over the sets of nodes and edges of a graph.

A program written in a certain language can be transformed into a syntax tree by a parser. When additional information such as bindings are included in the representation, the syntax tree is extended into a *syntax graph*. One main contribution of this report is to define a type graph model for syntax graphs that represent programs written in Java. The type graph model is complete, i.e., it covers the entire language specification up to version 1.6 [Sun]. We believe that this model can be of interest to any graph-based technique that relies on a representation of Java programs as graphs. As one example, suppose a visual programming/modelling tool that generates Java code from a graph; this could for instance, be used in the context of graph transformation-based model transformation [ALP07] or code refactoring [BGK01]. By enforcing the graph to be an instance of this type graph model, the tool can generate syntactically correct code.

The main challenge of our task it that we are dealing with a real imperative programming language, which has a complex definition. Thus, the creation of a comprehensive type graph for it is far from a trivial task.

In our current research we aim to perform static analysis [NNH99] and model checking [BK08] of Java programs using GROOVE [Ren03], a tool for state space exploration where states are represented as graphs, and the transitions from one state to another are given by graph transformation rules. A syntax graph is the static representation of a program as a graph, and it is the required initial structure for the subsequent elaboration of the states that constitute the dynamic behavior of the program. Thus, the work here presented is the first necessary step in our planned approach for the verification of code.

The visualization of a syntax graph can usually become verbose and confusing, even for small fragments of input code. Although we present syntax graphs as pictures in this document, it should be stressed that we do not intend to use syntax graphs as visual representations of programs (in fact, the source code is much more convenient for this purpose). Instead, what we intend to do is to use the generated syntax graphs as data structures, that can then be programmatically manipulated by the GROOVE tool without user intervention.

The rest of this report is structured as follows. The necessary definitions and concepts are presented in Chapter 2; in Chapter 3 the approach taken to elaborate the type graph model and to generate syntax graphs is explained. In Chapter 4 the type graph is presented and discussed. Finally, a comparison with related work is given in Chapter 5, and some conclusions and future work are listed in Chapter 6.

Chapter 2

Preliminaries and core concepts

In this chapter we will introduce the concepts of graphs, in particular type graphs and instance graphs. We do this on an informal level; this can easily be formalised, for instance using the approach outlined in [KR08].

Throughout, we assume the existence of a set of labels, which includes the special labels Boolean, Double, Float, Integer, Long, and String. These six special labels are used to define the types of attributes that can appear on a graph (see the last item of this chapter, on next page). We first present the definition of a graph.

Definition 1 (Graph) A *graph* consists of a set of nodes, a set of edges, and source and target functions from the edges to the nodes. Graphs will be depicted by displaying the nodes as boxes and the edges as arrows from their source nodes to their target nodes.

We use two kinds of graphs: type graphs and instance graphs. The nodes and edges of a type graph are used as node types and edge types (also called *associations*) of the instance graphs: every instance graph has a morphism to a type graph, which associates a node type with each of its nodes and an edge type with each of its edges (which are then sometimes called node and edge instances, respectively).

Definition 2 (Type Graph) A type graph is a graph with the following additional structure:

- Every node and edge has an associated unique *label* (or *name*). This is also taken to be the label of the respective instance nodes and edges;
- There is a binary acyclic *inheritance* relation over the nodes. The idea is that the node type of the source of each individual edge instance is smaller or equal, according to this inheritance relation, than the source of its edge type;
- A subset of the edge types are marked as *composition edges*. Such edges encode a "part-of" relation: the target node instances are considered to be *part of* the source node instances with which they are connected. Because a node cannot be part of more than one other node, the composition edge instances should form a tree within the instance graph.
- A subset of the composition edge types are marked as *ordered*. For all such edge types, there should be a total ordering among the target node instances that are connected by these edge instances to (and hence are part of) any given source node instance. This ordering will be encoded by adding integer index attributes to the target nodes, which contain the sequence number within the ordered list.
- Every edge has an associated *multiplicity*, which is a range $i..j$ of natural numbers with $i \leq j$, or such that $j = *$. The multiplicity indicates how many edge instances there should exist for every individual source node instance, where $*$ stands for an unbounded number.

Graphically, we use the following conventions (mostly following the UML notations):

- Edge instances are labelled with the name of the corresponding edge types;

- Node instances are labelled with the name of the corresponding node types *and all its supertypes*;
- Inheritance is denoted by open-ended arrows;
- Composition edge types are distinguished by black diamonds at their source ends;
- Ordered edge types are distinguished by the denotation {ordered} at their target ends;
- Multiplicities are denoted at the target ends of the edges; the multiplicity 1..1 is denoted 1, and in the absence of any denotation the implicit multiplicity $0..*$ is assumed;
- Attributes are not denoted as edges; rather, they are included in their respective source nodes as "name:Type" in type graphs, and as "name = value" in instance graphs, where "name" is the edge type label, "Type" the target node type label (e.g., Boolean, etc), and "value" the target node instance.

Chapter 3

Description of approach taken

The task of constructing syntax graphs from given source code consists of two major steps, (i) the elaboration of a type graph model to represent the syntactical elements of the chosen programming language, and (ii) the development of a tool that constructs a valid syntax graph from syntactically correct code. A syntax graph is considered to be valid when it is an instance of the type graph model developed in step (i). Essentially, the work to be done in (ii) boils down to writing a compiler that produces a syntax graph as its target language, instead of machine code.

In order to decide which is the more adequate approach for solving the presented task, we give a list of requirements that ideally should be fulfilled.

1. **The approach should be comprehensive and systematic.** We want an approach that allows us to elaborate the type graph in an organized and systematic way, such that in the end the process yields a model that covers the whole language.

2. **The approach should aim for automation.** Manual execution of steps is tedious and error-prone. Although we do not expect that a solution for the presented task can be fully automated, we want to keep manual intervention to a minimum.

3. **The approach should not try to reinvent the wheel.** Since the construction of a compiler for a real programming language is a quite complex and time-consuming task, we would like to reuse available tools and components as much as possible.

4. **The approach should be flexible.** The approach taken for a certain programming language should be, to some extent, reproducible for similar languages. Furthermore, the solution should be modular, in the sense that changes in the language syntax can be handled by local adjustments and do not force us to start again from scratch. Finally, we would like to have some freedom of choice on the representation of the type graph.

Among the possible solutions that were considered but later discarded we can cite:

- **Elaboration of the type graph from the language specification.** Although flexible, this approach was considered too ad-hoc to be of interest.

- **Construction of a compiler from scratch.** This was the approach taken in previous research where a proof-of-concept case was presented for a toy programming language [KKR06]. Although feasible for simple languages, this approach is unattractive when working with a language that has complex syntax and semantics, such as Java.

- **Use of available compiler parts.** There are some Java grammars[1] and parsers[2] that could be used for the compiler construction, however some steps of code analysis such as name and type resolution would still have to be done manually, a work that we deemed unnecessary. Furthermore, most of these artifacts do not receive much maintenance and therefore are not kept up to date with the language specification.

[1] `http://www.antlr.org/grammar/list`
[2] `http://code.google.com/p/javaparser/`

After discarding the approaches above, we decided to adapt an open-source Java compiler for our purposes. In doing so, the implementation effort is kept to a minimum, since we have only to modify the code generation phase of the compiler to construct the syntax graphs. Also, by analysing the source code of the compiler we are able to elaborate the type graph model in a very straightforward way. Thus, with this solution, the definition of the type graph and the construction of the syntax graph generator go hand in hand, and we have the guarantee that a syntax graph generated from code is compliant with the type graph model.

One possible drawback of this approach is loss of freedom in the graphs representation. By using a specific compiler we are somewhat restricted by its structure. However, given the benefits of the approach we consider this to be an acceptable compromise over the requirements.

3.1 Creating the type graph

In order to develop our chosen approach we decided to use the Eclipse Java Compiler [Ecl]. This compiler is also written in Java, and its source code is available for use under the Eclipse Public License. The compiler source is divided in several packages, among which the package `org.eclipse.jdt.internal.compiler.ast`[3] is of particular interest, since it is where the classes that compose the Abstract Syntax Tree (AST) built by the compiler are grouped. By analysing the package contents we are able to construct the type graph model, which is presented in Chapter 4.

The `ast` package contains, for example, classes like `Expr` and `Stmt`[4] to represent expressions and statements of the Java language. In fact, every syntactical element of the language has a corresponding class in the `ast` package and those classes are grouped in a certain hierarchy. The top most class is `ASTNode`, which defines a common super type for all elements of the AST. The `ast` package also provides an AST visitor pattern interface [GHJV95], which has methods to navigate over the nodes of the AST in a depth-first-like manner.

```
class IfStmt extends Stmt {
  // fields
  Expr condition;
  Stmt thenStmt;
  Stmt elseStmt;
  ...
  // AST traversal method
  public void traverse(ASTVisitor v) {
    condition.traverse(v);
    thenStmt.traverse(v);
    if (elseStmt != null)
      elseStmt.traverse(v);
    ...
  }
  ...
}
```

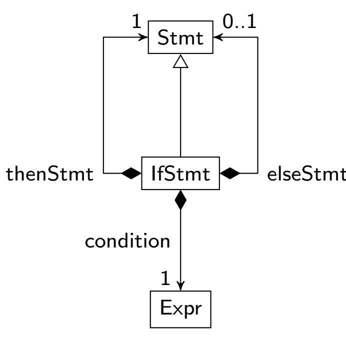


Figure 3.1: Example of the type graph elaboration from the compiler source code

The way the type graph is elaborated from the elements of the `ast` package can be better explained with an example. Figure 3.1 shows the relevant code of the class that represents an “if” statement and the corresponding part of the type graph constructed from this code. We start with the class name, `IfStmt`, that gives rise to an homonymous node type in the type graph. Also, since `IfStmt` is a subclass of `Stmt` we create another node type for the super class and we insert an inheritance relation in the type graph, between the corresponding node types. The class fields that are references to other classes of the AST become compositions (in some cases, ordered ones) in the type graph, with labels matching the field names. In this example, the fields named `condition`, `thenStmt`, and `elseStmt` give rise to three compositions in the type graph, with corresponding labels. Additionally, the way the visitor pattern is

[3]Through the rest of the report we adhere to the following convention: elements of Java code are shown in `typewriter` font, while elements of the type graph or instance graphs are shown in sans serif font.

[4]In fact, these are abbreviations of the name of the classes in the compiler source code. We do this change here for presentation purposes, in order to save space. The full list of abbreviations used is given in Figure 4.1.

implemented in the class provides some guidance over the cardinalities of the compositions just created. From the implementation of the `traverse` method we see that fields `condition` and `thenStmt` are always visited. Therefore we can conclude that the IfStmt node type must have mandatory condition and thenStmt compositions, a fact that is illustrated by the cardinality 1 of those compositions in the type graph. On the other hand, the check for non-nullness of the `elseStmt` field indicates that it may not always exists. Therefore we mark the cardinality of its composition in the type graph as 0..1.

By analysing the classes of interest of the `ast` package in the same way as described in the example above we can elaborate a large part of the type graph model. This provides us with the comprehensive and systematic approach for type graph construction that we sought. However, there are some elements of the type graph that still need to be manually created. As an example we can cite the associations that resolve name and type references, which correspond to the binding edges on syntax graphs. The intuition for identifying where these associations must be created is simple: any reference should have an association with a corresponding declaration; however, the information needed to create these associations is not present in the compiler source code in a uniform way, and therefore manual intervention is necessary. The rationale behind our decisions over what does or does not have to be manually inserted into the type graph comes from our intended purpose for the syntax graphs. Thus, we insert only the elements that we deem necessary for static analysis and simulation.

3.2 Constructing syntax graphs from code

To construct syntax graphs from Java code we must change the back end of the Eclipse Java Compiler. By stopping the compiler after parsing and code analysis but before machine code generation we are able to profit from the work done by the compiler until this stage. Specifically, name and type references are already resolved, simplifying the construction of the syntax graph.

We developed a syntax graph generator that implements the AST visitor interface provided by the compiler and we plugged it in the compiler back end. To build the syntax graph, our generator visits the AST, performing the following steps.

- For each node in the AST the generator creates a corresponding node in the syntax graph. The types of a syntax graph node are obtained through reflection. By using reflection in Java, one is able to query the virtual machine for run-time information of objects. In our case we obtain the class hierarchy of an AST node via reflection and store this information as a label of the syntax graph node.

- For the construction of edges in the syntax graph we keep an auxiliary mapping of AST nodes into syntax graph nodes. This mapping, along with the bindings produced by the compiler, is sufficient for creating the edges, including the ones that resolve references.

For each node type of the type graph we created a test case input program. With these test cases we can inspect the syntax graphs produced by our tool and check for implementation errors. An example of such test case is given in Section 4.3, along with the corresponding syntax graph generated. The complete set of input test cases is given in Appendix A.

Chapter 4

Type graph model

In this section we present the type graph model for the executable constructs of the Java programming language. Language elements that do not have an effect on the execution of the program, such as comments and annotations, were deliberately left out. As stated in Chapter 1, our goal is to perform a simulation of the execution of Java programs, a task for which only the executable elements of the language need to be considered. Apart from that the type graph model covers the whole language specification up to version 1.6 [Sun].

Given the size of the model, we present it in parts for ease of understanding. First we present the class diagram of the type graph, showing the hierarchy of node types. The associations between node types and their attributes are given subsequently, in separate diagrams. Due to space limitations, it is not possible to display all node types of the type graph in this chapter. We consider the ones shown in this section as good representatives of the relations in the model. A complete list of the node types is given in Appendix A.

4.1 Class diagram

The class diagram of the type graph is shown in Figure 4.1. It is formed by 75 node types, mapped directly from the compiler classes. The 13 nodes with labels in italic correspond to abstract classes in the compiler.

One interesting abstract node type is Stmt, which not only has several concrete subtypes representing a variety of language constructs, but also has four other abstract subtypes, including Expr. This unusual relation between expressions and statements comes from the compiler implementation. From a semantic point of view, some expressions in Java, e.g., assignments, may be used as statements. In the compiler source code the designers explain that in order to avoid the creation of wrappers around expressions that are used as statements, they decided to make Expr a subclass of Stmt and let the parser handle incorrect usage of an expression as a statement. It is important to note that given the approach taken for its construction, our type graph model is bound to inherit the design decisions made by the crafters of the chosen compiler.

The Ref node type is also worthy of note. It has different node subtypes to denote references to array declarations (ArrayRef), field declarations, local declarations, and to instances of objects (this, super). The unusual fact that super is a subtype of this also comes from the compiler design. These incongruences with regard to the Java language specification are correctly dealt with by the compiler in its static analysis phase, however we will still have to take them into account on future steps of our work, for example when defining the semantics of the elements of a syntax graph in terms of graph transformation rules. We regard this as minor adjustments that do not additionally complicate the required work for the next steps.

4.2 Node types, attributes and associations

Figure 4.2 depicts the type declaration node type, which represents a class or interface declaration in Java. From the figure we see that a type declaration has a name attribute and is composed of zero or

Abbreviations:

Abs – Abstract

Alloc – Allocation

Assign – Assignment

Comp – Compilation

Cons – Constructor

Decl – Declaration

Expr – Expression

Lit – Literal

Qual – Qualified

Ref – Reference

Stmt – Statement

Var – Variable

Figure 4.1: Class diagram of the type graph model

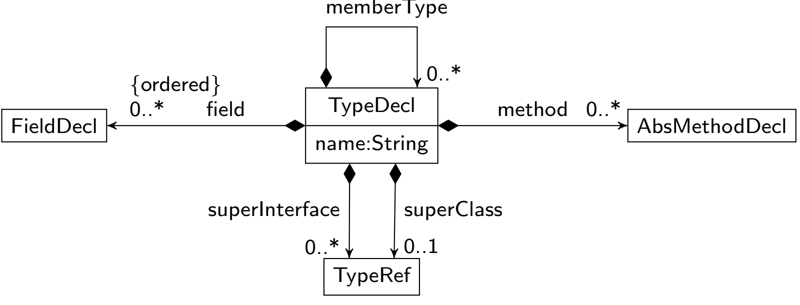


Figure 4.2: Type declaration

more method declarations and zero or more field declarations (those are present only when the type declaration represent a class, interfaces cannot have fields.) The field composition is marked as {ordered} due to the fact that fields that appear later in a class declaration can have initialization expressions that refer to previously declared fields. Thus, during the construction of an object, its fields must be created on the same order of their declaration in the class. Additionally, to represent inheritance, the type declaration node type has two compositions with type references. A class can implement zero or more interfaces (superInterface composition) and can extend at most one class. The cardinality of the superClass composition is explained by the fact that only explicitly declared inheritances are mapped by this composition. Type declarations that are a direct subclass of the Java class Object do not need a superClass composition. Finally, the memberType composition represents the Java concept of nested classes.

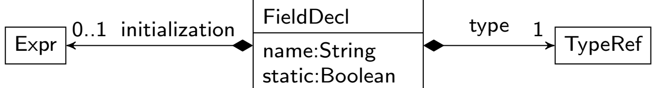


Figure 4.3: Field declaration

A field declaration is shown in Figure 4.3. It has a name attribute, a mandatory composition with a type reference, and an optional initialization expression. It also has a Boolean attribute, to indicate if the field is declared to be static. There is no need to represent others field modifiers, such as public, private, etc., because they are statically checked by the compiler and do not have influence on program execution. The choice of which field modifiers should be included on the type graph was manual. It is a good example of specific language characteristics that need to be individually analysed and therefore hinder the creation of a fully automated process for the type graph generation.

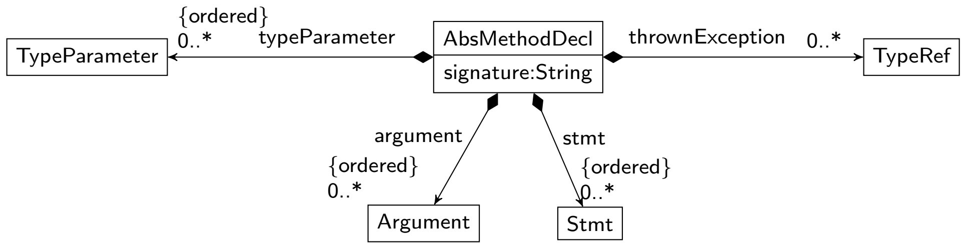


Figure 4.4: Abstract method declaration

Figure 4.4 presents an abstract method declaration, which is an abstract node type. Although it can only exist as either a method declaration or a constructor declaration, the common attributes and compositions of both these concrete node types are summarized within the abstract method declaration node type. From the figure we see that every method declaration is formed by zero or more arguments (usually called formal parameters) and zero or more statements. The signature attribute is composed by the method name and the types of the arguments. Furthermore, a method declaration can indicate which exceptions it can throw, and can have a list of type parameters if the method is generic.

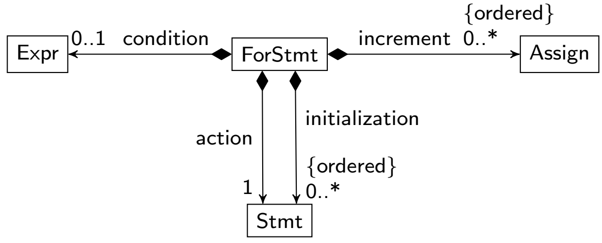


Figure 4.5: For statement

The representation of a for statement is given in Figure 4.5. It is composed by a list of initialization

statements, a non-mandatory condition expression, an action statement that is either a simple statement or a Block of statements, and a list of increment assignments to the loop variables.

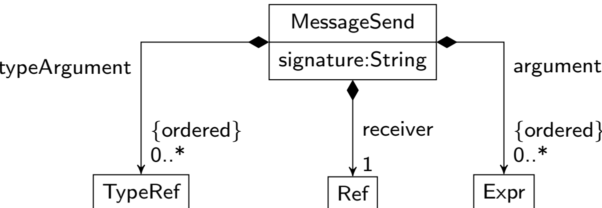


Figure 4.6: Message send

Figure 4.6 shows the static structure used to represent a method call. It is not possible to provide a static binding of method calls due to Java polymorphism. Instead, those bindings have to be resolved during run-time. A MessageSend has a mandatory receiver, which is a reference to the object that triggered the call, a signature attribute, that is used to resolve the call to a method declaration, and a list of arguments, that constitute the real parameters that will be matched to the formal parameters in the method declaration. A method call may also have a list of type arguments, if the invoked method is generic.

Figure 4.7: Name reference

Figure 4.7 depicts the node type for name references. It is always possible to statically determine to which variable declaration a name reference is bound. It is important to note that the refersTo association is not a composition. This is due to the fact that one variable declaration may have an arbitrary number of references. The node type for type references is quite similar to this one.

4.3 Example of a syntax graph

In order to illustrate how some of the node types and associations presented in Section 4.2 may appear in a syntax graph, we conclude this section with an example of a syntax graph that is an instance of the type graph model.

Figure 4.8 presents a small piece of Java code and its corresponding syntax graph. The description on how such syntax graphs are constructed from source code is given in Section 3.2. The syntax graph in Figure 4.8 has a node labeled TypeDecl, that corresponds to the node type shown in Figure 4.2. The name of the declared class is stored as an attribute of the node, which also has three outgoing edges that correspond to the field and method compositions. The nodes labeled FieldDecl were automatically given an extra index attribute to cope with the requirement that field edges must be {ordered}. The syntax graph also has a node labeled MethodDecl which is a subtype of the node type given in Figure 4.4. Finally, it is important to note that name and type reference nodes have an outgoing edge that binds the reference to its corresponding declaration. We consider the existence of a "system" compilation unit, where the primitive types of the language, and also the classes of `java.lang.*`, are declared. Part of this "system" compilation unit is shown in the bottom of Figure 4.8 (within the dashed box), with the declaration of the primitive type `int`.

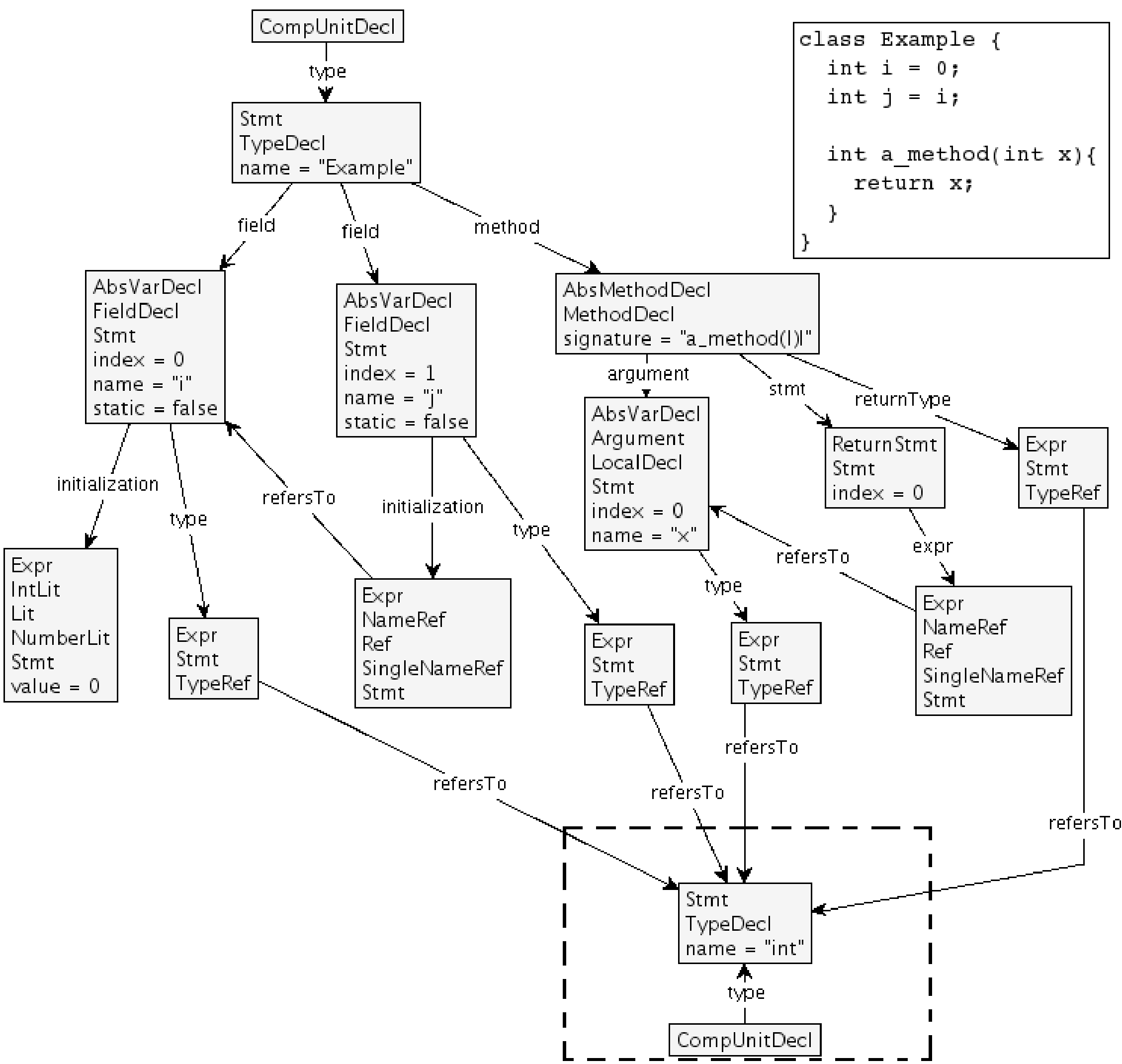


Figure 4.8: Example of a syntax graph built from code

Chapter 5

Related work

Essentially, every compiler uses an internal data type representation of compiled programs, which in some way encodes the abstract syntax graph in the sense of this report. The purpose of this report is to make this representation explicit and to model it as a typed graph. We believe this aim to be more or less new; here we briefly list what little related work we could find.

First of all, the OMG meta-model for Java [Obj04] has a clear correspondence with the type declaration part of our type graph (see Figure 4.1 and Figure 4.2). However, this meta-model “stops” at the executable level: it does not go below method declarations. Furthermore, it appears that the standardisation effort is so big that this definition suffers from the “maintenance” problem discussed in Chapter 3: the OMG meta-model is for Java 1.3, and no newer version seems to be forthcoming. A project with just the same aim, documented in [DMSS01], seems to have suffered the same fate.

The idea of generating type and instance graphs by using an existing Java grammar (rather than an existing compiler) was explored in [AP04] and demonstrated on a subset of Java. The motivation there was to enable model transformation rather than verification. Unfortunately the work was never published.

Closest in spirit to our work is [CDFR04], that define a semantics for Java on the basis of graph transformations. A large part of that paper is devoted to the definition of a type graph and the corresponding instance graphs; however, this is entirely a manual effort, in that no attempt was made to use an existing grammar. No doubt partially as a consequence of this, the fragment of Java covered is relatively small.

As for our planned work for the verification of programs, there are other different approaches with the same intent, some already quite mature. A work similar to ours is the Java PathFinder (JPF) project [MGPMS] which is a software model checker for Java byte-code. The two main differences to our intended research are:

- **The input language.** Whereas JPF works on compiled Java byte-code, we work with Java source code. In doing so we believe that our approach can be more easily adapted to other programming languages.
- **The program state representation.** JPF is an explicit state model-checker that relies on the Java Virtual Machine for the generation of the state space, hence the representation of a state of the program is just a snapshop of the virtual machine state. In our work we are defining program states as graphs, and we want to investigate how abstract interpretation techniques can be used to simplify these states. In doing so we believe that the problem of state space explosion, always present in model-checking techniques, can be mitigated.

An alternative approach for software verification is theorem proving. Among the tools developed under this approach we can cite the KeY System [BHS07] and Why/Krakatoa [FM07] for the verification of Java programs, and the Spec# tool [BRLS04], that analyses code written in a super set of the C# language. In particular, these tools produce verification conditions that are then discharged using automated SMT solvers or interactive theorem provers. The main drawback of these tools is the need for code instrumentation, usually on the form of annotations in the input code. Most of the software produced to date lack this sort of instrumentation and the required effort for its elaboration is very high. Although many annotations can be automatically generated from the code, some, such as loop invariants, still need to be provided by the user.

Chapter 6

Conclusion and future work

To sum up, the contributions of this report are threefold.

- We have presented a comprehensive type graph that covers all executable elements of the Java programming language. Such type graph can be of interest as a model for tools that represent Java programs as graphs.
- We have shown a straightforward and systematic approach for the elaboration of the type graph model by analysing a compiler source code. Although our described method focused on Java, we believe that it can be adapted (with varying degrees of difficulty) to other programming languages as well.
- We explained how the back end of a compiler can be adapted in order to automatically construct a syntax graph representation from source code.

The work described in this report is the first step in our planned approach for the verification of Java programs. Now that we are able to generate syntax graphs from code the next step is the construction of *flow graphs*, structures that model the sequential execution relation between elements of the syntax graph. We plan to define graph transformations rules over syntax graphs for flow graph construction, as described in [KKR06]. Together, a syntax graph and a flow graph form a *program graph.* The subsequent step is then use the GROOVE tool to simulate the execution of program graphs. Another important aspect of this step is that we want to apply abstract interpretation techniques to simplify the program graphs and thus improve the performance of the simulation.

Bibliography

[ALP07] Marcus Alanen, Torbjörn Lundkvist, and Ivan Porres, *Creating and reconciling diagrams after executing model transformations*, Sci. Comput. Program. **68** (2007), no. 3, 155–178.

[AP04] Marcus Alanen and Ivan Porres, *A relation between context-free grammars and meta object facility metamodels*, Tech. report, TUCS – Turku Centre for Computer Science, March 2004.

[BGK01] Dirk Baumer, Erich Gamma, and Adam Kiezun, *Integrating refactoring support into a Java development tool*, OOPSLA 2001 Companion, 2001.

[BHS07] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt (eds.), *Verification of object-oriented software: The KeY approach*, LNCS 4334, Springer-Verlag, 2007.

[BK08] C. Baier and J. P. Katoen, *Principles of model checking*, MIT Press, New York, May 2008.

[BRLS04] Mike Barnett, K. Rustan, M. Leino, and Wolfram Schulte, *The Spec# programming system: An overview*, CASSIS 2004, LNCS vol. 3362, Springer, 2004, pp. 49–69.

[CDFR04] Andrea Corradini, Fernando Luís Dotti, Luciana Foss, and Leila Ribeiro, *Translating Java code to graph transformation systems*, International Conference on Graph Transformations (ICGT) (Hartmut Ehrig, Gregor Engels, Francesco Parisi-Presicce, and Grzegorz Rozenberg, eds.), Lecture Notes in Computer Science, vol. 3256, Springer, 2004, pp. 383–398.

[DMSS01] Jan Docks, Kristof Mertens, Nele Smeets, and Eric Steegmans, *jnome: A Java meta model in detail*, Report CW 232, Katholieke Universiteit Leuven, December 2001, URL: `http://www.cs.kuleuven.be/~marko/jnome/index.html`.

[Ecl] Eclipse Foundation, *JDT core component development resources*, `http://www.eclipse.org/jdt/core/dev.php`.

[FM07] Jean-Christophe Filliâtre and Claude Marché, *The Why/Krakatoa/Caduceus platform for deductive program verification*, CAV (Werner Damm and Holger Hermanns, eds.), Lecture Notes in Computer Science, vol. 4590, Springer, 2007, pp. 173–177.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design patterns: Elements of reusable object-oriented software*, Addison-Wesley Professional Computing Series, Addison-Wesley Publishing Company, New York, NY, 1995.

[KKR06] Harmen Kastenberg, Anneke Kleppe, and Arend Rensink, *Defining object-oriented execution semantics using graph transformations*, Formal Methods for Open Object-Based Distributed Systems (FMOODS) (Roberto Gorrieri and Heike Wehrheim, eds.), Lecture Notes in Computer Science, vol. 4037, Springer, 2006, pp. 186–201.

[KR08] A. G. Kleppe and A. Rensink, *On a graph-based semantics for UML class and object diagrams*, Graph Transformation and Visual Modelling Techniques, Budapest, Hungary (C. Ermel, J. De Lara, and R. Heckel, eds.), Electronic Comminucations of the EASST, vol. 10, EASST, 2008.

[MGPMS] Peter Mehlitz, Dimitra Giannakopoulou, Corina Pasareanu, and Masoud Mansouri-Samani, *Java PathFinder*, `http://javapathfinder.sourceforge.net/`.

[NNH99] Flemming Nielson, Hanne R. Nielson, and Chris Hankin, *Principles of program analysis*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[Obj04] Object Management Group, *Metamodel and UML profile for Java and EJB specification*, Tech. Report formal/04-02-02, version 1.0, OMG, February 2004, URL: `http://www.omg.org/docs/formal/04-02-02.pdf`.

[Ren03] Arend Rensink, *The GROOVE simulator: A tool for state space generation*, Applications of Graph Transformations with Industrial Relevance (AGTIVE) (John L. Pfaltz, Manfred Nagl, and Boris Böhlen, eds.), Lecture Notes in Computer Science, vol. 3062, Springer, 2003, pp. 479–485.

[Sun] Sun Microsystems, *The Java language specification*, `http://java.sun.com/docs/books/jls/`.

Appendix A

Comprehensive list of node types of the type graph

- This appendix presents all the 75 node types of the type graph shown in Figure 4.1. The node types are given in alphabetical order.
- The node types associations and attributes are presented in an incremental way, meaning that information already given in super types is not repeated in the subtypes. For ease of reference, in such cases relevant links to other pages of the appendix are given in the comments part of the node type. Some abstract node types do not have any additional information whatsoever. They are only included in this appendix for the sake of completeness.
- We do not show the "system" compilation unit in each example and by doing so it is not possible to draw the corresponding refersTo associations in the syntax graph. Instead, the TypeRef nodes are shown with an additional string attribute resolvedType that indicates which type of the "system" compilation unit is being referenced. This change was made for presentation purposes only.

Node type	ABSTRACT METHOD DECLARATION

TypeParameter
{ordered}
0..*
typeParameter
AbsMethodDecl
signature:String
thrownException
0..*
TypeRef
argument
{ordered}
0..*
Argument
stmt
{ordered}
0..*
Stmt

Java code example

Corresponding syntax graph

Comments

- This node type is abstract. See also pages 37, 41, and 65 for the concrete subtypes of this node type.
- This node type was explained in detail on Figure 4.4.

Node type	ABSTRACT VARIABLE DECLARATION

AbsVarDecl

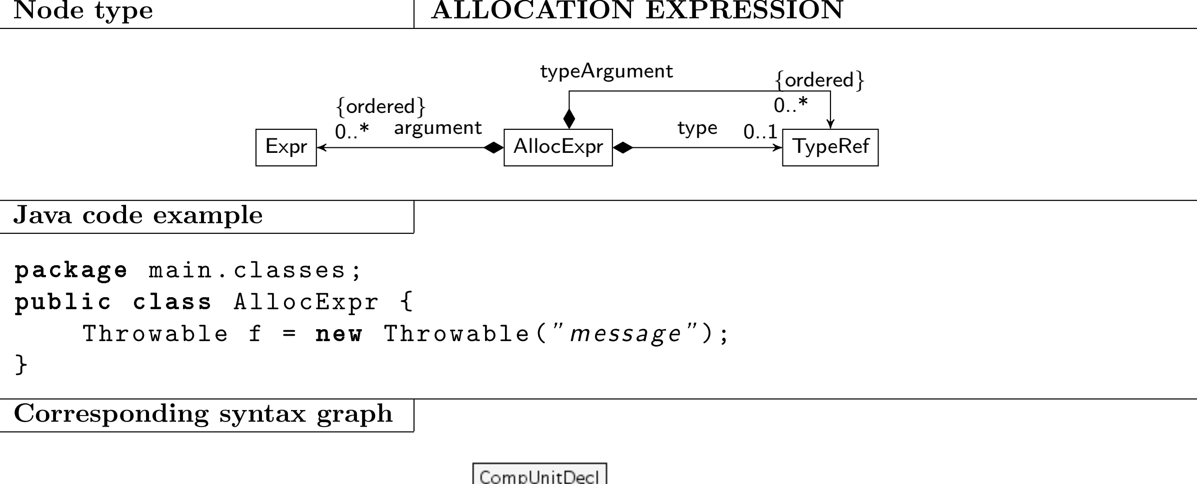
Java code example

Corresponding syntax graph

Comments

- This node type is abstract. See also pages 22, 50, 56, 61, and 90 for the concrete subtypes of this node type.

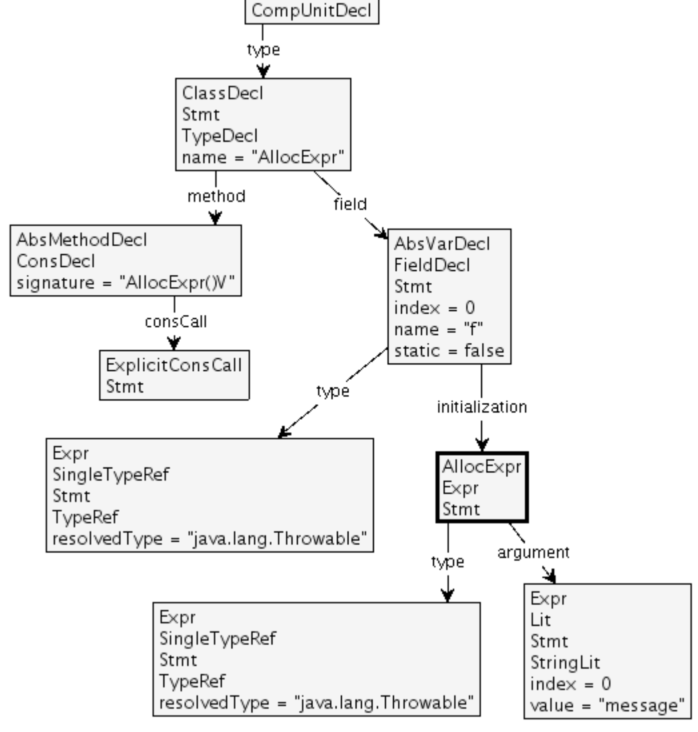
Node type	ALLOCATION EXPRESSION



Java code example

```
package main.classes;
public class AllocExpr {
    Throwable f = new Throwable("message");
}
```

Corresponding syntax graph



Comments

- This node type represents expressions with the reserved word `new` that do not involve arrays. The type reference identifies the type of the object that will be created and the `argument` associations are the parameters for the object constructor.

Node type	ARGUMENT

Argument

Java code example

```
package main.classes;
public class Argument {
    public void method (int i) {
    }
}
```

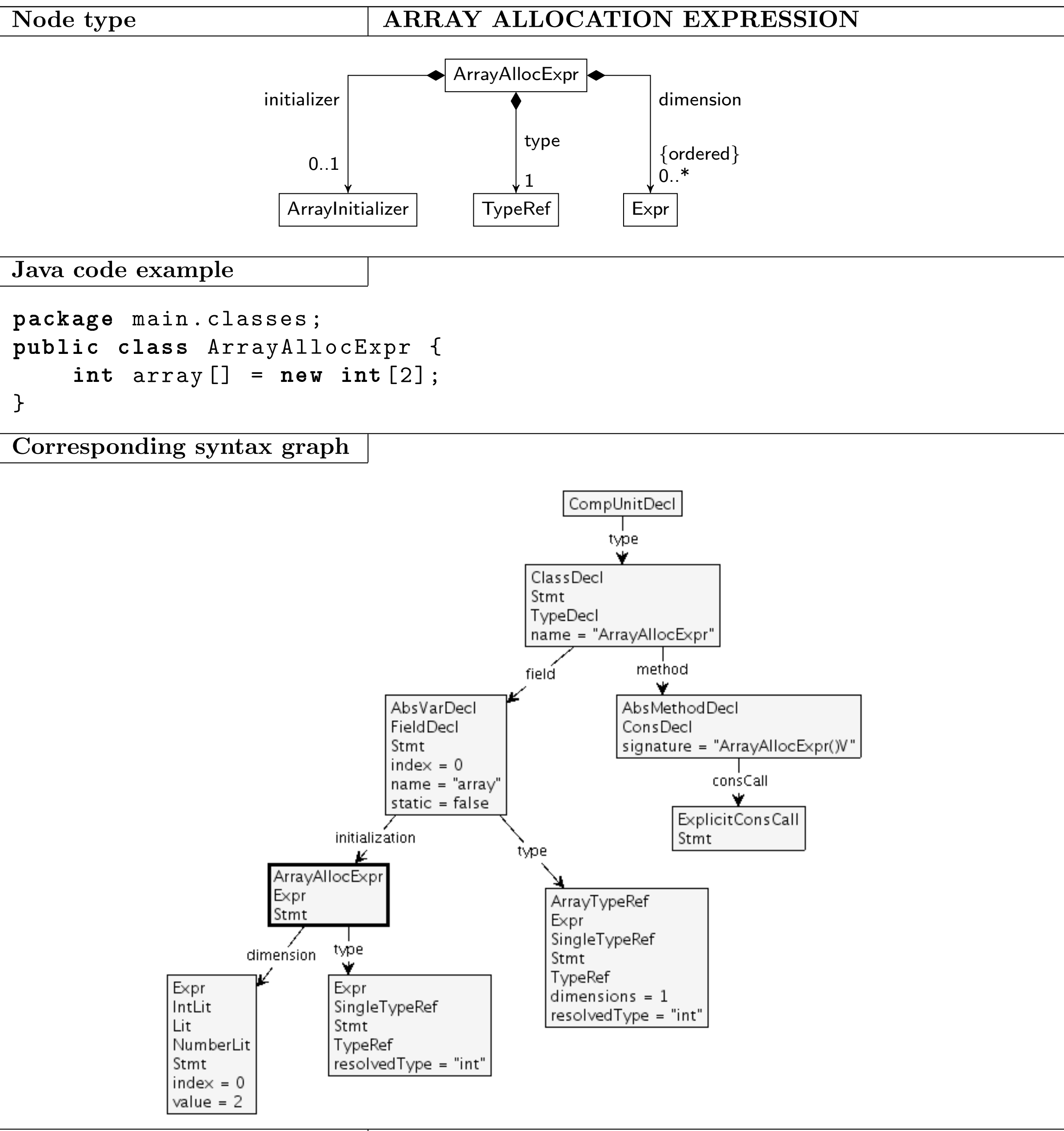
Corresponding syntax graph

CompUnitDecl
type →
ClassDecl / Stmt / TypeDecl / name = "Argument"
method → AbsMethodDecl / ConsDecl / signature = "Argument()V"
consCall → ExplicitConsCall / Stmt
method → AbsMethodDecl / MethodDecl / signature = "method(I)V"
returnType → Expr / SingleTypeRef / Stmt / TypeRef / resolvedType = "void"
argument → AbsVarDecl / Argument / LocalDecl / Stmt / index = 0 / name = "i"
type → Expr / SingleTypeRef / Stmt / TypeRef / resolvedType = "int"

Comments

- See the super type on page 61 for additional information.

Node type	ARRAY ALLOCATION EXPRESSION



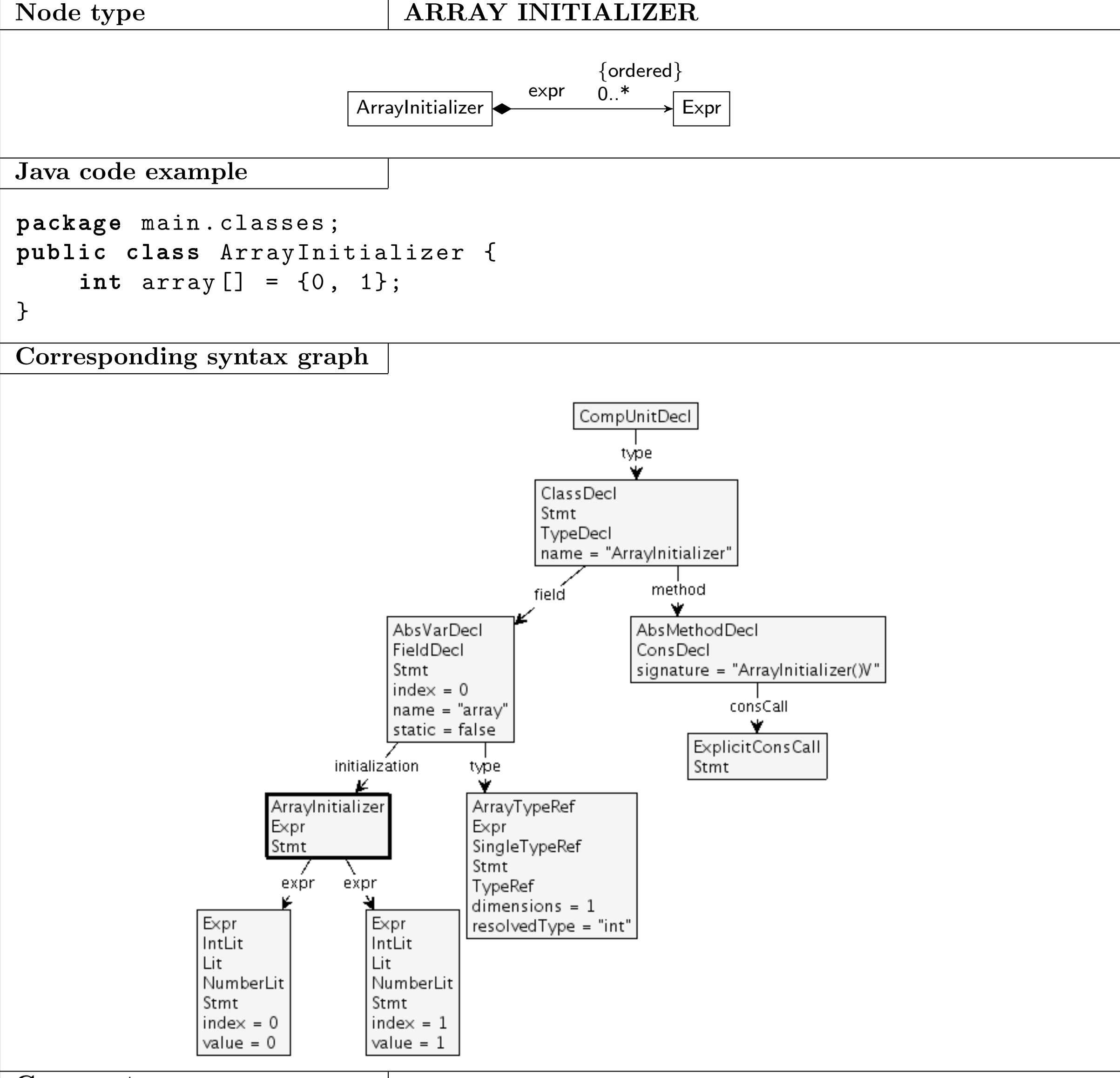
Java code example

```java
package main.classes;
public class ArrayAllocExpr {
    int array[] = new int[2];
}
```

Corresponding syntax graph

Comments

- This node type represents expressions with the reserved word `new` that dynamically allocate arrays. The dimension associations can represent multi-dimensional arrays.



Node type	ARRAY INITIALIZER

Java code example

```
package main.classes;
public class ArrayInitializer {
    int array[] = {0, 1};
}
```

Corresponding syntax graph

Comments

- This node type represents a list of expressions that are used to initialize the elements of an array. Array initializers can be nested when using multi-dimensional arrays.

Node type	ARRAY REFERENCE

Ref 1 receiver ArrayRef position 1 Expr

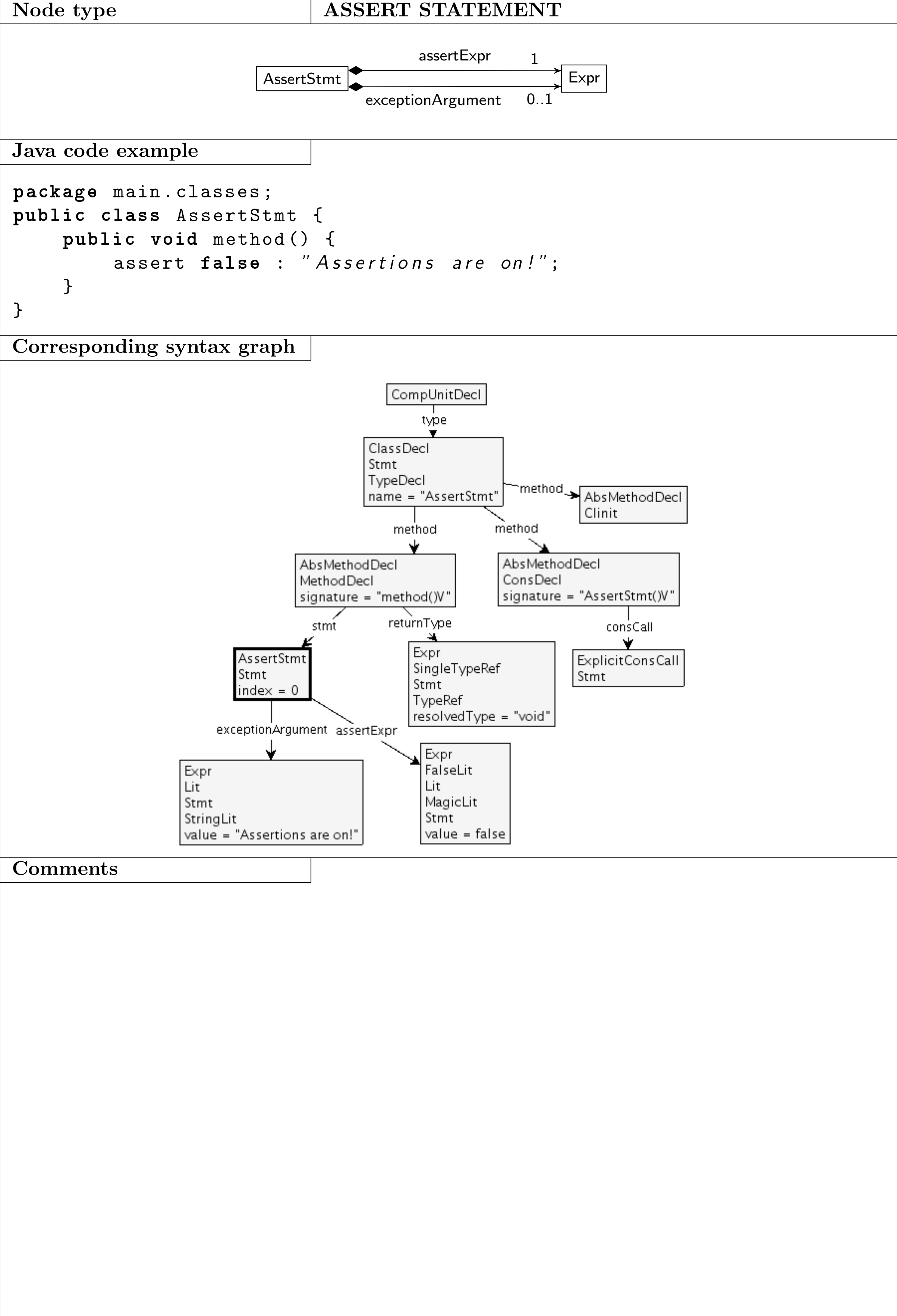
Java code example

```
package main.classes;
public class ArrayRef {
    public void method() {
        int array[] = new int[42];
        array[0] = 0;
    }
}
```

Corresponding syntax graph

Comments

- This node type represents a reference to a specific element of an array.



Node type	ASSERT STATEMENT

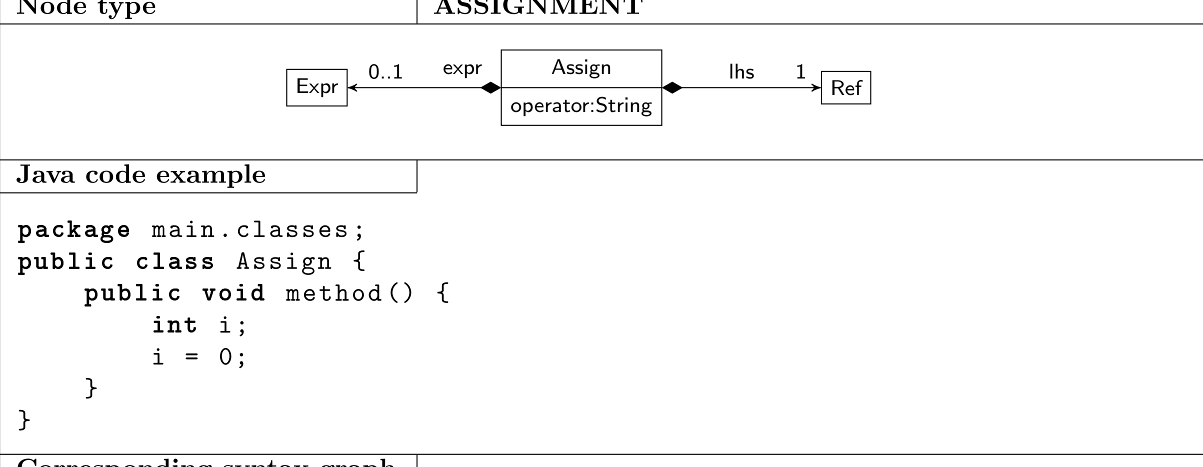
Java code example

```
package main.classes;
public class AssertStmt {
    public void method() {
        assert false : "Assertions are on!";
    }
}
```

Corresponding syntax graph

Comments

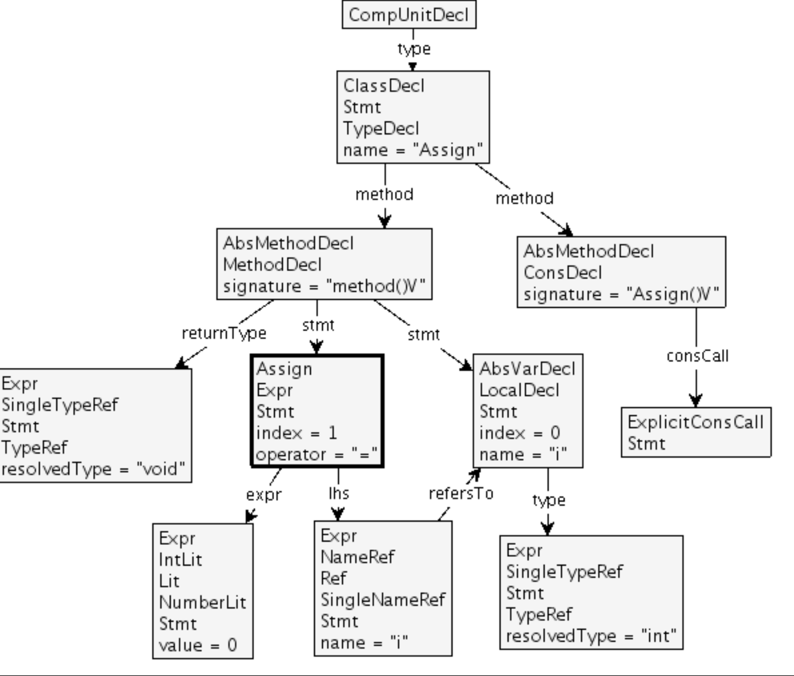
Node type	ASSIGNMENT



Java code example

```java
package main.classes;
public class Assign {
    public void method() {
        int i;
        i = 0;
    }
}
```

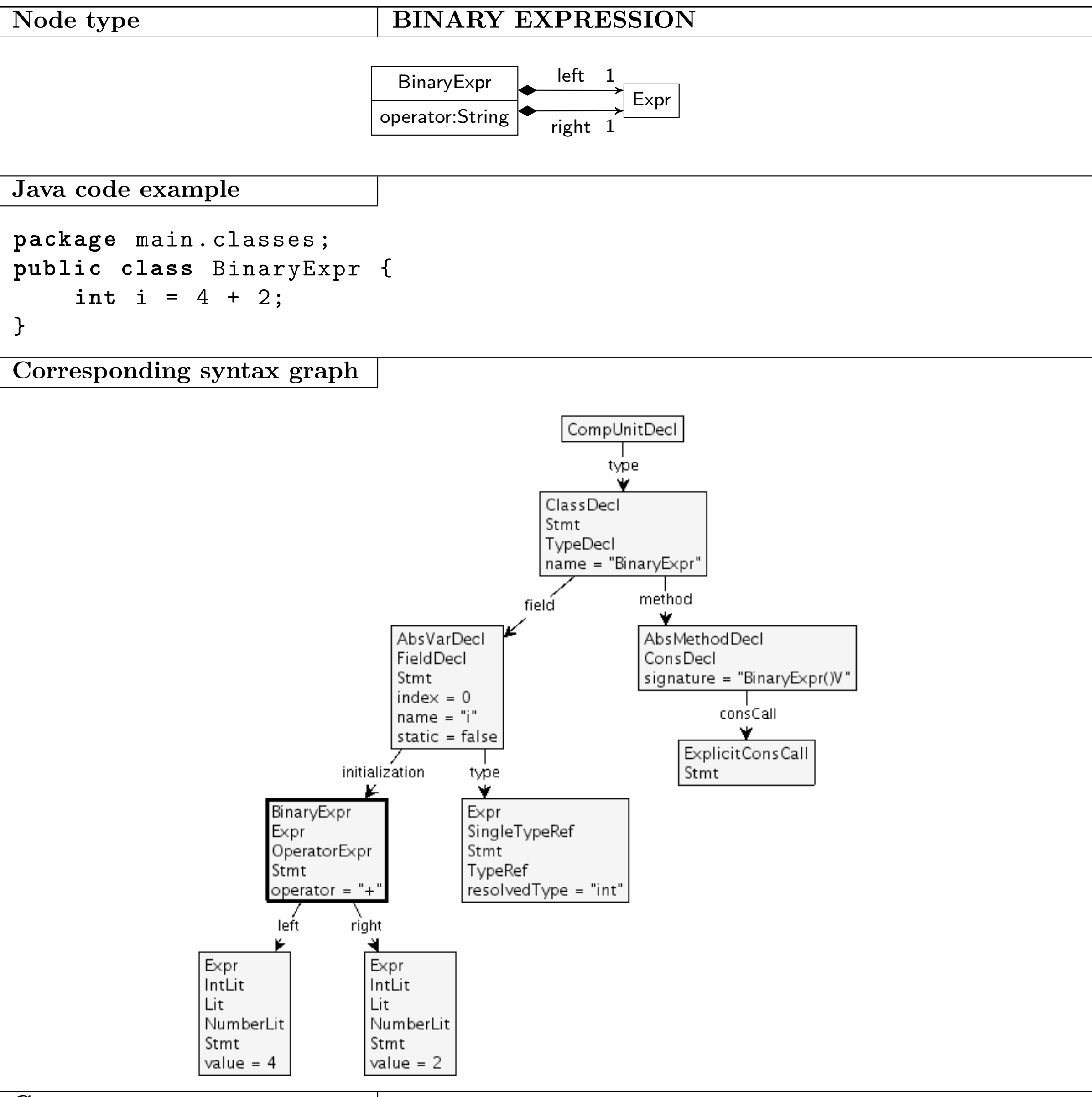
Corresponding syntax graph



Comments

- This node type has an operator attribute to store the operator of compound assignments, such as +=.
- See also pages 39, 70, and 71 for subtypes of this node type.

Node type	AST NODE
ASTNode	
Java code example	
Corresponding syntax graph	
Comments	
• This node type is abstract and is the base of the whole node types hierarchy.	



Comments

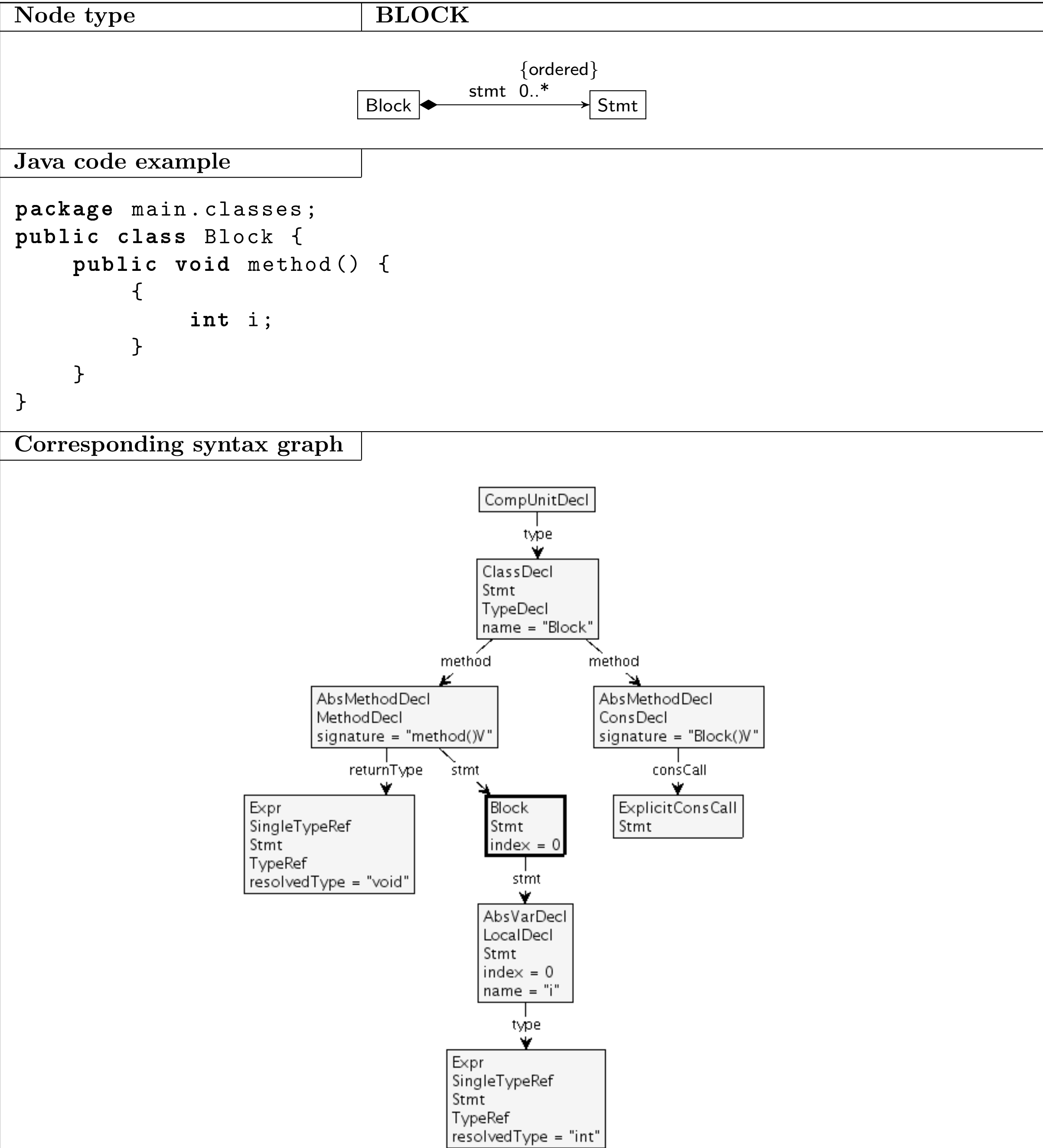
- This node type represents expressions such as `&&`, `*`, etc.

Node type	BLOCK

Java code example

```
package main.classes;
public class Block {
    public void method() {
        {
            int i;
        }
    }
}
```

Corresponding syntax graph



Comments

- An empty Block (with zero statements) is different than an empty statement (;). See also page 45.

Node type	BRANCH STATEMENT

BranchStmt —label 0..1→ LabeledStmt

Java code example

Corresponding syntax graph

Comments

- This node type is abstract. See also pages 32, and 42 for the concrete subtypes of this node type.
- It is important to note that the label association is not a composition.

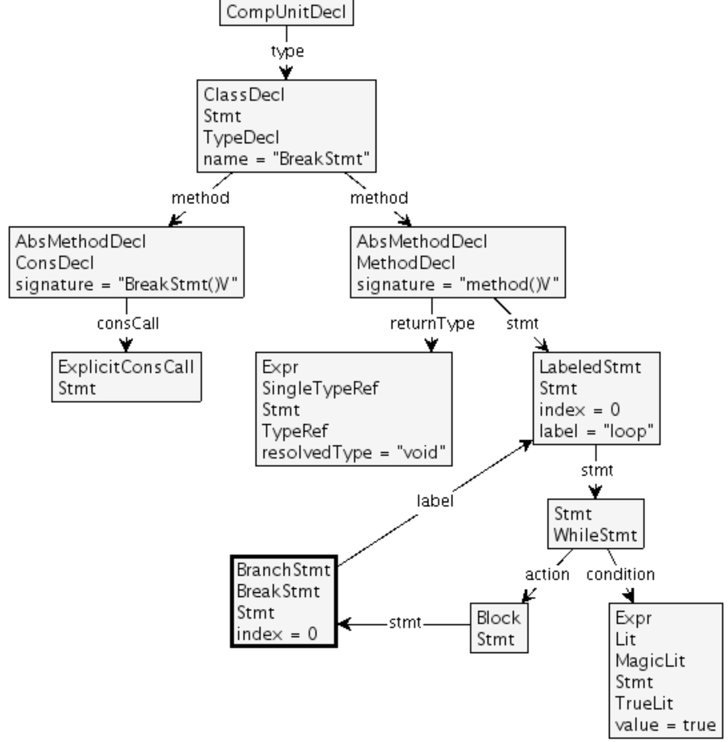
Node type	BREAK STATEMENT

BreakStmt

Java code example

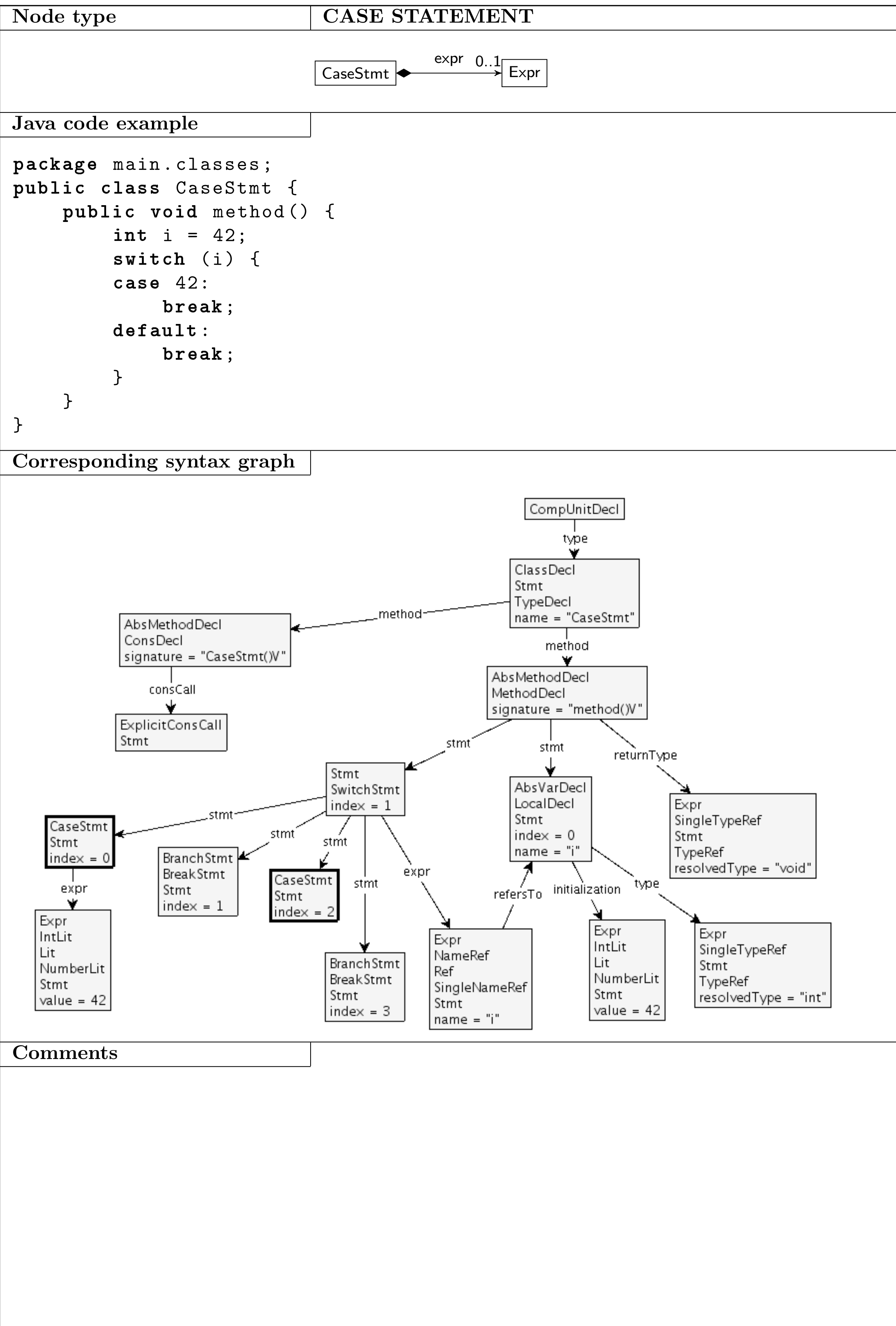
```java
package main.classes;
public class BreakStmt {
    public void method() {
        loop:
        while (true) {
            break loop;
        }
    }
}
```

Corresponding syntax graph



Comments

- See the super type on page 31 for additional information.



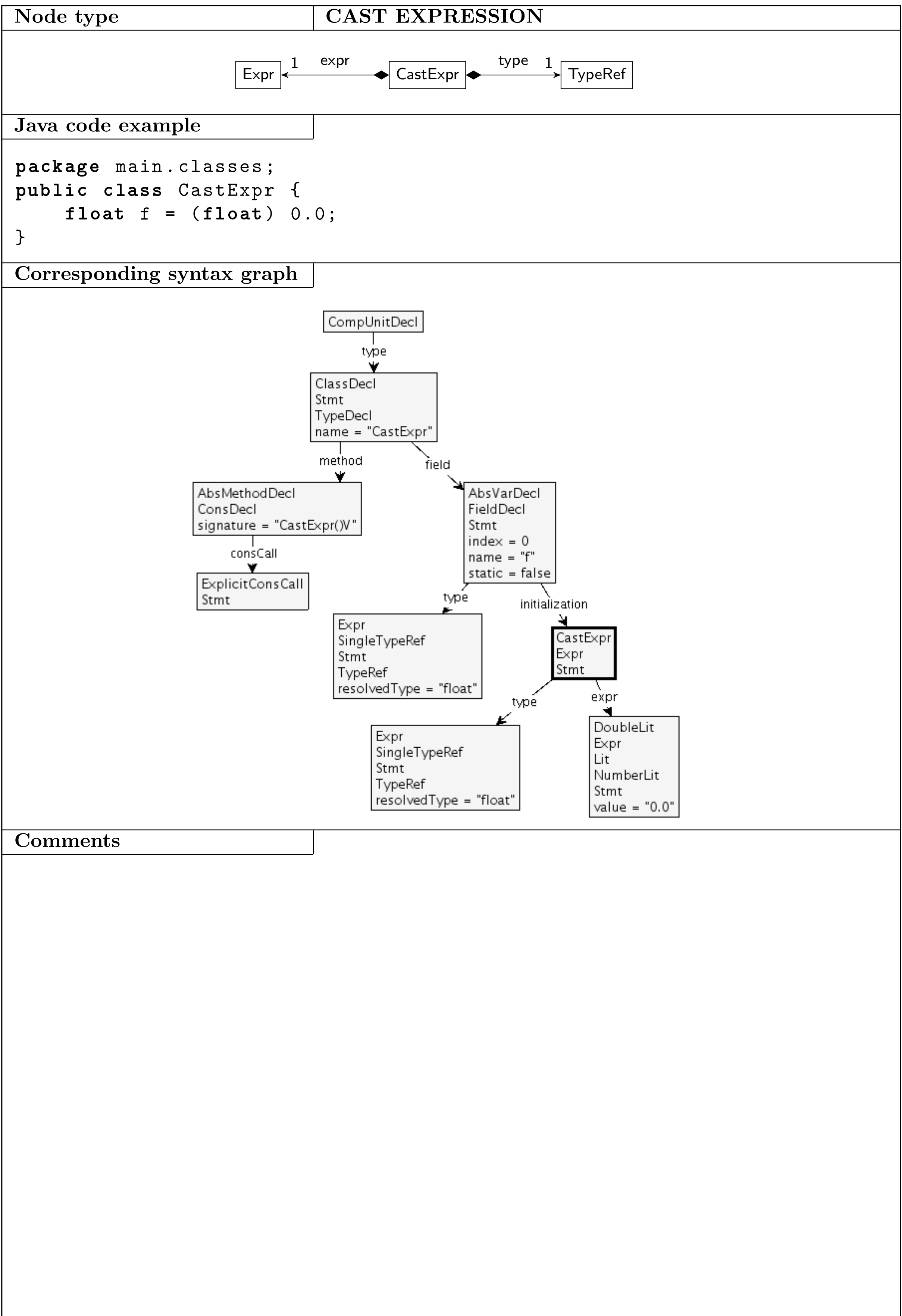
Node type	CASE STATEMENT

Java code example

```java
package main.classes;
public class CaseStmt {
    public void method() {
        int i = 42;
        switch (i) {
        case 42:
            break;
        default:
            break;
        }
    }
}
```

Corresponding syntax graph

Comments


Node type
CAST EXPRESSION
Expr
1
expr
CastExpr
type
1
TypeRef
Java code example
package main.classes;
public class CastExpr {
float f = (float) 0.0;
}
Corresponding syntax graph
CompUnitDecl
type
ClassDecl
Stmt
TypeDecl
name = "CastExpr"
method
field
AbsMethodDecl
ConsDecl
signature = "CastExpr()V"
consCall
ExplicitConsCall
Stmt
AbsVarDecl
FieldDecl
Stmt
index = 0
name = "f"
static = false
type
initialization
Expr
SingleTypeRef
Stmt
TypeRef
resolvedType = "float"
CastExpr
Expr
Stmt
type
expr
Expr
SingleTypeRef
Stmt
TypeRef
resolvedType = "float"
DoubleLit
Expr
Lit
NumberLit
Stmt
value = "0.0"
Comments

Node type	CHAR LITERAL

CharLit
value:String

Java code example

```
package main.classes;
public class CharLit {
    char c = 'a';
}
```

Corresponding syntax graph

CompUnitDecl

type

ClassDecl
Stmt
TypeDecl
name = "CharLit"

method

field

AbsMethodDecl
ConsDecl
signature = "CharLit()V"

consCall

ExplicitConsCall
Stmt

AbsVarDecl
FieldDecl
Stmt
index = 0
name = "c"
static = false

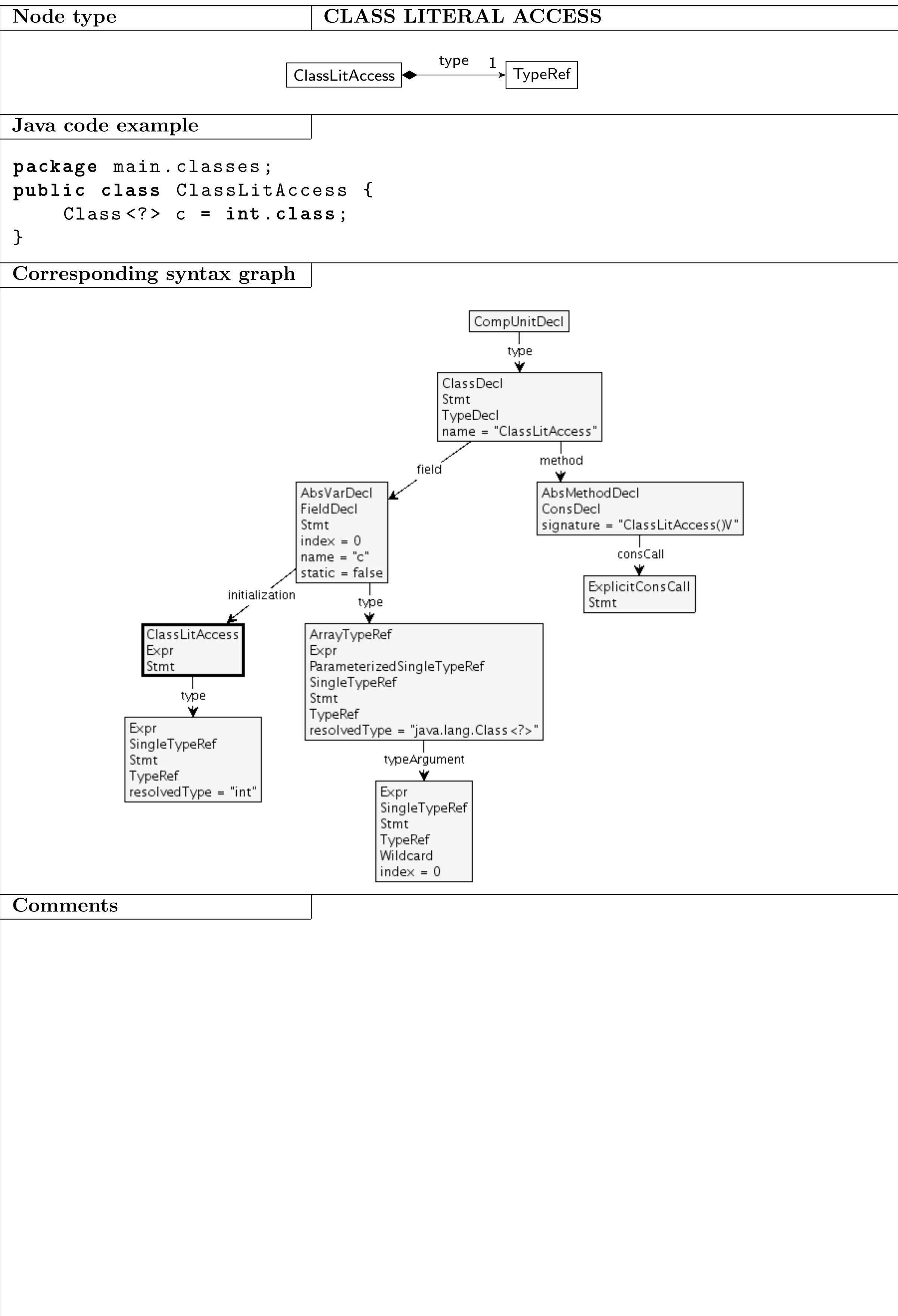
initialization

type

CharLit
Expr
Lit
NumberLit
Stmt
value = "a"

Expr
SingleTypeRef
Stmt
TypeRef
resolvedType = "char"

Comments


Node type
CLASS LITERAL ACCESS
ClassLitAccess
type
1
TypeRef
Java code example
package main.classes;
public class ClassLitAccess {
Class<?> c = int.class;
}
Corresponding syntax graph
CompUnitDecl
type
ClassDecl
Stmt
TypeDecl
name = "ClassLitAccess"
field
method
AbsVarDecl
FieldDecl
Stmt
index = 0
name = "c"
static = false
AbsMethodDecl
ConsDecl
signature = "ClassLitAccess()V"
consCall
ExplicitConsCall
Stmt
initialization
type
ClassLitAccess
Expr
Stmt
type
Expr
SingleTypeRef
Stmt
TypeRef
resolvedType = "int"
ArrayTypeRef
Expr
ParameterizedSingleTypeRef
SingleTypeRef
Stmt
TypeRef
resolvedType = "java.lang.Class<?>"
typeArgument
Expr
SingleTypeRef
Stmt
TypeRef
Wildcard
index = 0
Comments

Node type	CLINIT

Clinit

Java code example

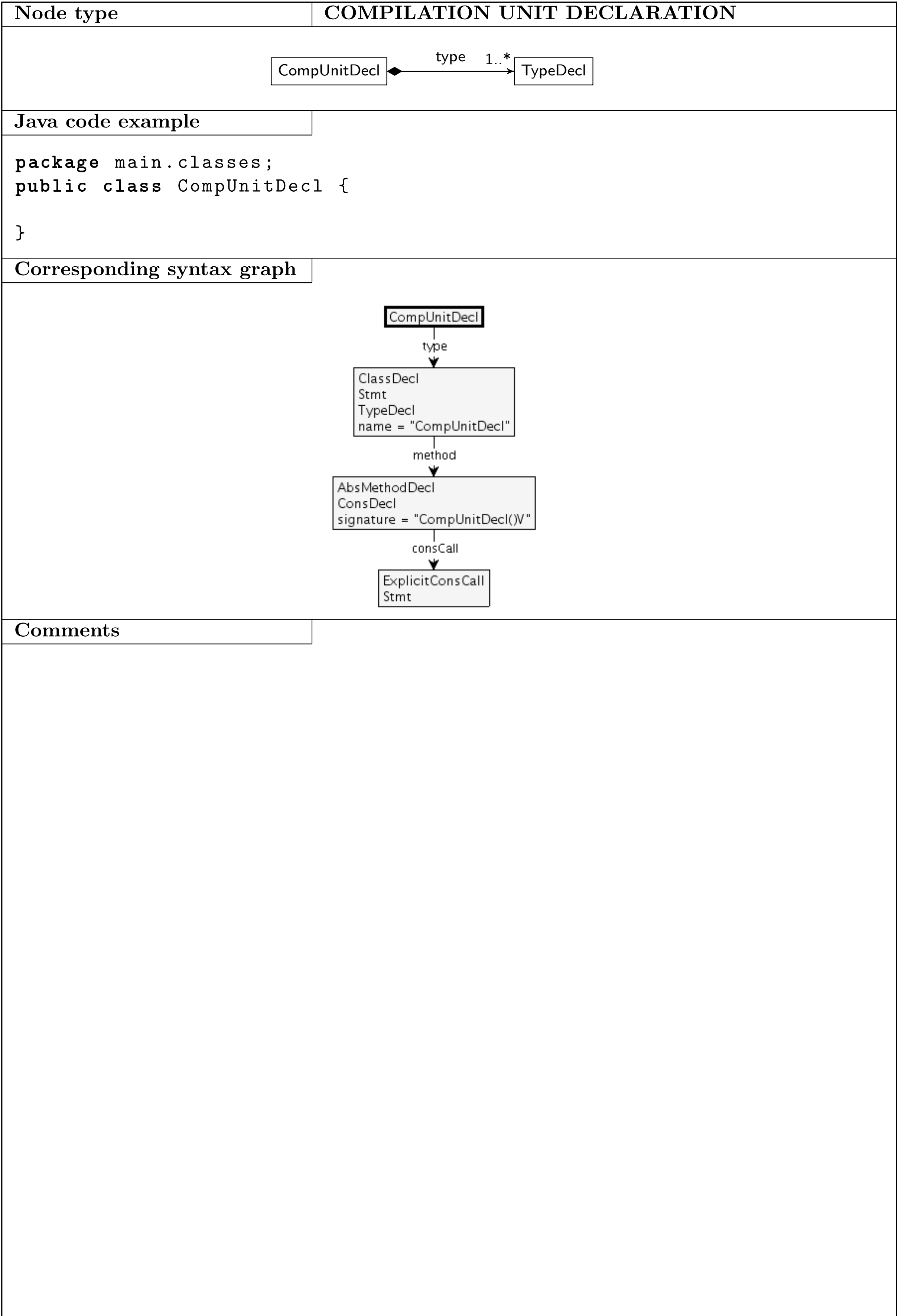
```
package main.classes;
public class Clinit {
    public static int i;
}
```

Corresponding syntax graph

CompUnitDecl
type
ClassDecl
Stmt
TypeDecl
name = "Clinit"
method
field
method
AbsMethodDecl
ConsDecl
signature = "Clinit()V"
AbsVarDecl
FieldDecl
Stmt
index = 0
name = "i"
static = true
AbsMethodDecl
Clinit
consCall
ExplicitConsCall
Stmt
type
Expr
SingleTypeRef
Stmt
TypeRef
resolvedType = "int"

Comments

- See the super type on page 19 for additional information.


Node type
COMPILATION UNIT DECLARATION
CompUnitDecl
type
1..*
TypeDecl
Java code example
package main.classes;
public class CompUnitDecl {
}
Corresponding syntax graph
CompUnitDecl
type
ClassDecl
Stmt
TypeDecl
name = "CompUnitDecl"
method
AbsMethodDecl
ConsDecl
signature = "CompUnitDecl()V"
consCall
ExplicitConsCall
Stmt
Comments

Node type	COMPOUND ASSIGNMENT

CompoundAssign

Java code example

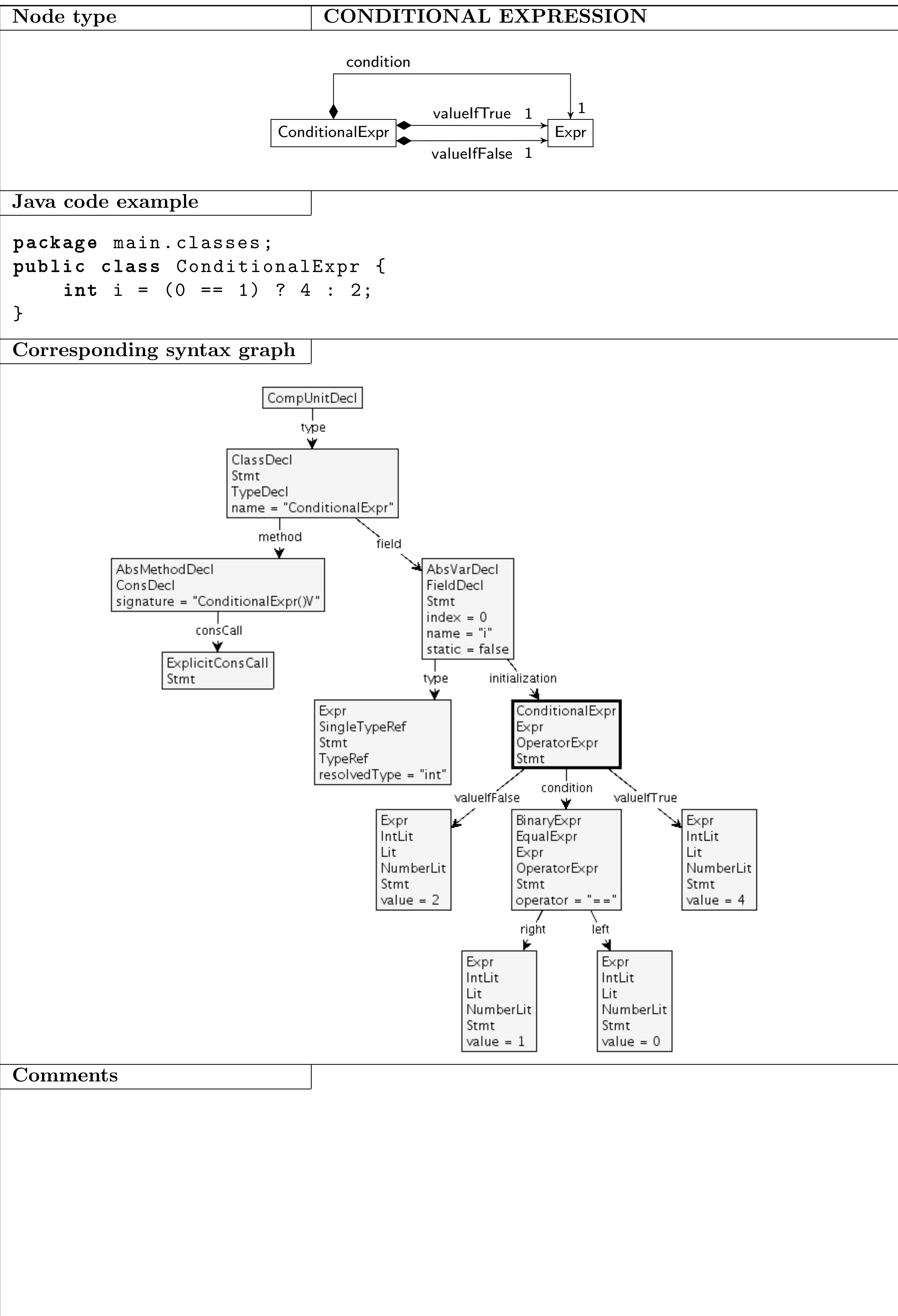
```java
package main.classes;
public class CompoundAssign {
    public void method() {
        int i = 0;
        i += 1;
    }
}
```

Corresponding syntax graph

CompUnitDecl
type
ClassDecl
Stmt
TypeDecl
name = "CompoundAssign"
method
method
AbsMethodDecl
MethodDecl
signature = "method()V"
AbsMethodDecl
ConsDecl
signature = "CompoundAssign()V"
returnType
stmt
stmt
consCall
Expr
SingleTypeRef
Stmt
TypeRef
resolvedType = "void"
AbsVarDecl
LocalDecl
Stmt
index = 0
name = "i"
Assign
CompoundAssign
Expr
Stmt
index = 1
operator = "+="
ExplicitConsCall
Stmt
type
initialization
refersTo
lhs
expr
Expr
SingleTypeRef
Stmt
TypeRef
resolvedType = "int"
Expr
IntLit
Lit
NumberLit
Stmt
value = 0
Expr
NameRef
Ref
SingleNameRef
Stmt
name = "i"
Expr
IntLit
Lit
NumberLit
Stmt
value = 1

Comments

- See the super type on page 27 for additional information.



Node type	CONDITIONAL EXPRESSION

Java code example

```
package main.classes;
public class ConditionalExpr {
    int i = (0 == 1) ? 4 : 2;
}
```

Corresponding syntax graph

Comments

Node type	CONSTRUCTOR DECLARATION

ConsDecl — consCall 0..1 → ExplicitConsCall

Java code example

```java
package main.classes;
public class ConsDecl {
    public ConsDecl(int i) throws Exception {
        boolean b;
    }
}
```

Corresponding syntax graph

CompUnitDecl
type
ClassDecl
Stmt
TypeDecl
name = "ConsDecl"
method
AbsMethodDecl
ConsDecl
signature = "ConsDecl(I)V"
thrownException
consCall
argument
stmt
Expr
SingleTypeRef
Stmt
TypeRef
resolvedType = "java.lang.Exception"
ExplicitConsCall
Stmt
AbsVarDecl
Argument
LocalDecl
Stmt
index = 0
name = "i"
AbsVarDecl
LocalDecl
Stmt
index = 0
name = "b"
type
Expr
SingleTypeRef
Stmt
TypeRef
resolvedType = "int"
type
Expr
SingleTypeRef
Stmt
TypeRef
resolvedType = "boolean"

Comments

- See the super type on page 19 for additional information.

Node type	CONTINUE STATEMENT

ContinueStmt

Java code example

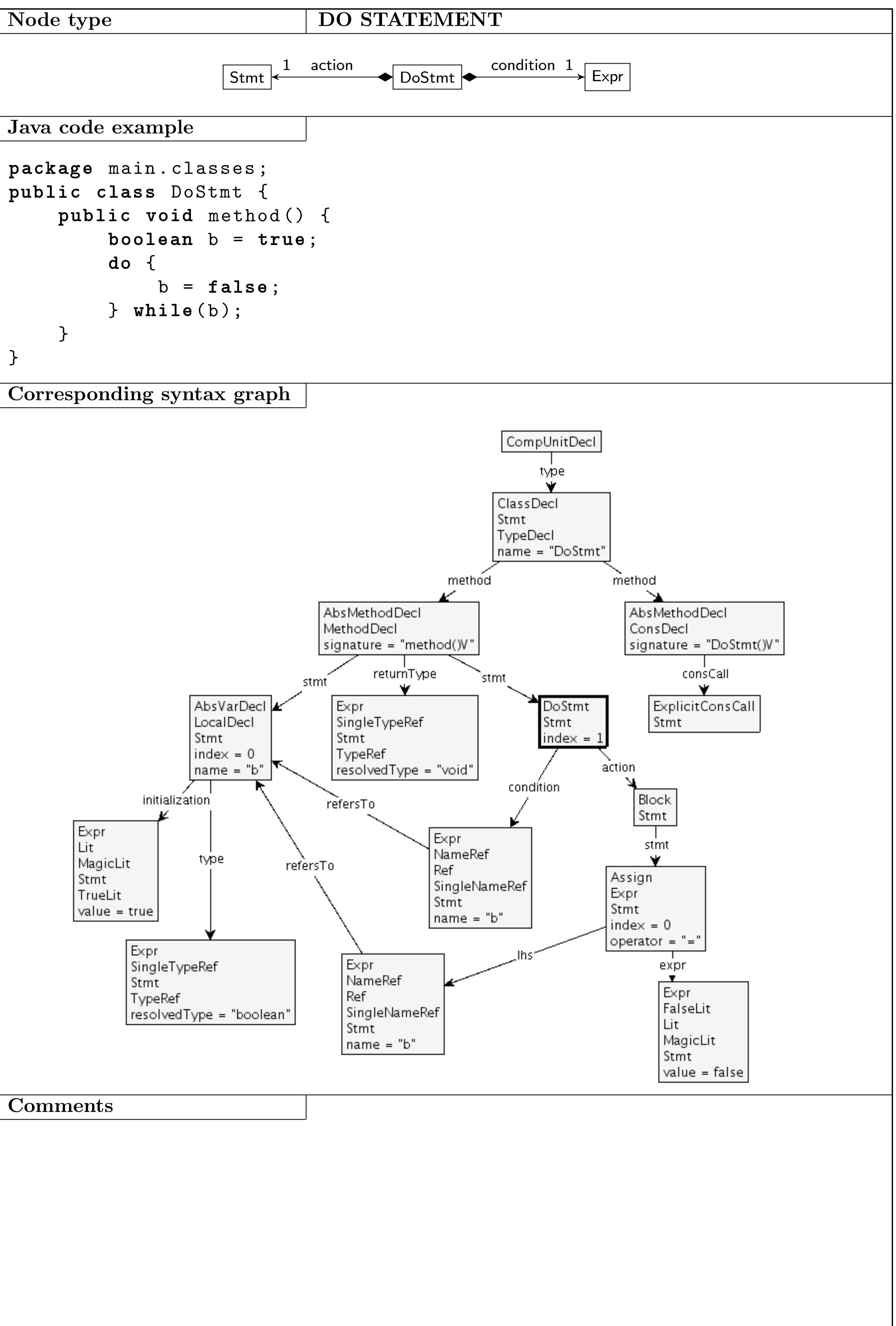
```
package main.classes;
public class ContinueStmt {
    public void method() {
        loop:
        while (true) {
            continue loop;
        }
    }
}
```

Corresponding syntax graph

CompUnitDecl
type
ClassDecl
Stmt
TypeDecl
name = "ContinueStmt"
method
method
AbsMethodDecl
MethodDecl
signature = "method()V"
AbsMethodDecl
ConsDecl
signature = "ContinueStmt()V"
stmt
returnType
consCall
LabeledStmt
Stmt
index = 0
label = "loop"
Expr
SingleTypeRef
Stmt
TypeRef
resolvedType = "void"
ExplicitConsCall
Stmt
label
stmt
Stmt
WhileStmt
action
condition
BranchStmt
ContinueStmt
Stmt
index = 0
stmt
Block
Stmt
Expr
Lit
MagicLit
Stmt
TrueLit
value = true

Comments

- See the super type on page 31 for additional information.


Node type
DO STATEMENT
Stmt
1
action
DoStmt
condition
1
Expr
Java code example
package main.classes;
public class DoStmt {
public void method() {
boolean b = true;
do {
b = false;
} while(b);
}
}
Corresponding syntax graph
CompUnitDecl
type
ClassDecl Stmt TypeDecl name = "DoStmt"
method
method
AbsMethodDecl MethodDecl signature = "method()V"
AbsMethodDecl ConsDecl signature = "DoStmt()V"
consCall
ExplicitConsCall Stmt
stmt
returnType
stmt
AbsVarDecl LocalDecl Stmt index = 0 name = "b"
Expr SingleTypeRef Stmt TypeRef resolvedType = "void"
DoStmt Stmt index = 1
condition
action
Block Stmt
initialization
refersTo
Expr Lit MagicLit Stmt TrueLit value = true
type
refersTo
Expr NameRef Ref SingleNameRef Stmt name = "b"
stmt
Assign Expr Stmt index = 0 operator = "="
lhs
expr
Expr SingleTypeRef Stmt TypeRef resolvedType = "boolean"
Expr NameRef Ref SingleNameRef Stmt name = "b"
Expr FalseLit Lit MagicLit Stmt value = false
Comments

Node type	DOUBLE LITERAL

DoubleLit
value:Double

Java code example

```java
package main.classes;
public class DoubleLit {
    double d = 1.0d;
}
```

Corresponding syntax graph

CompUnitDecl
type
ClassDecl
Stmt
TypeDecl
name = "DoubleLit"
method
field
AbsMethodDecl
ConsDecl
signature = "DoubleLit()V"
consCall
ExplicitConsCall
Stmt
AbsVarDecl
FieldDecl
Stmt
index = 0
name = "d"
static = false
initialization
type
DoubleLit
Expr
Lit
NumberLit
Stmt
value = "1.0"
Expr
SingleTypeRef
Stmt
TypeRef
resolvedType = "double"

Comments

Node type	EMPTY STATEMENT

EmptyStmt

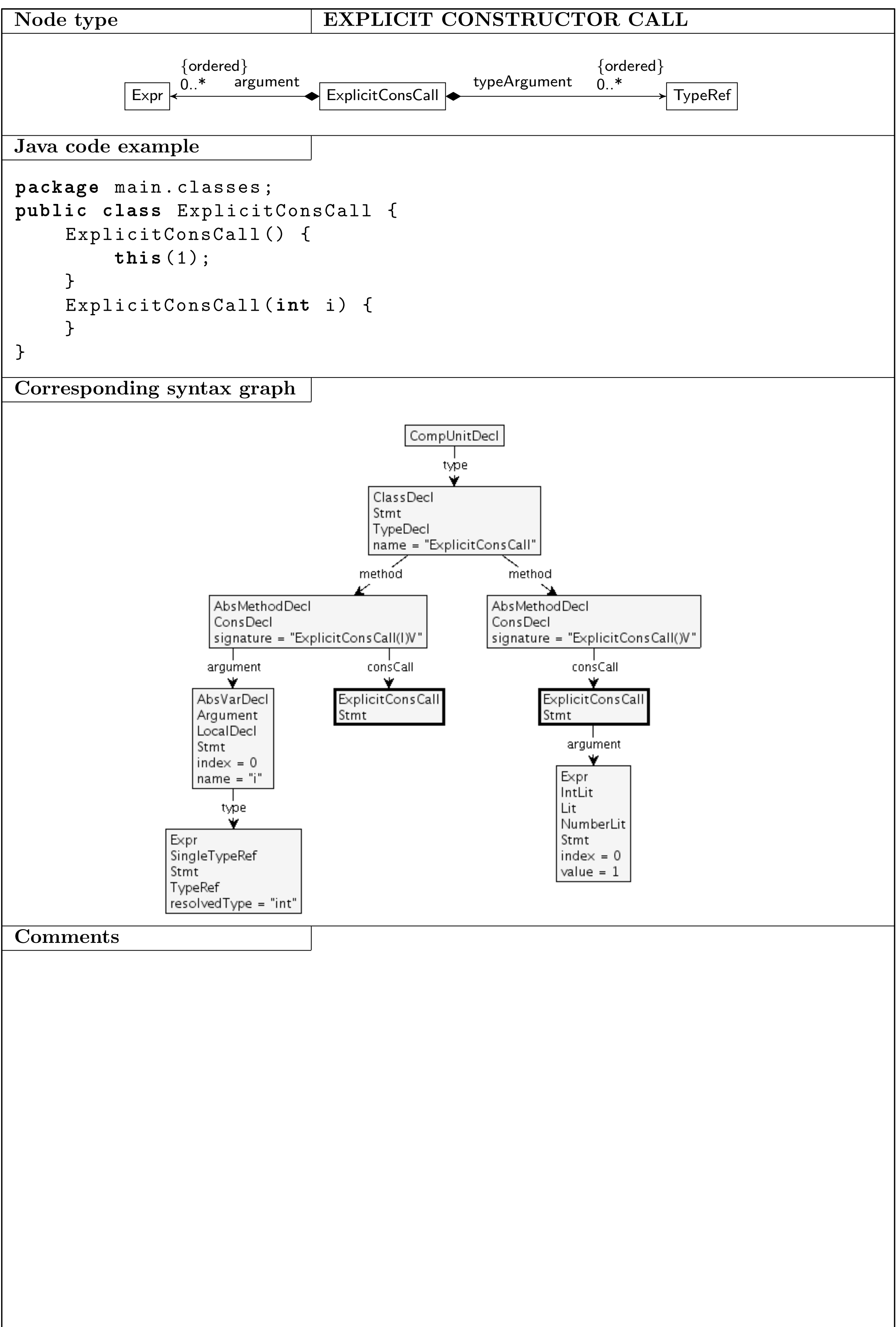
Java code example

```
package main.classes;
public class EmptyStmt {
    public void method() {
        ;
    }
}
```

Corresponding syntax graph

CompUnitDecl
type
ClassDecl
Stmt
TypeDecl
name = "EmptyStmt"
method
method
AbsMethodDecl
ConsDecl
signature = "EmptyStmt()V"
AbsMethodDecl
MethodDecl
signature = "method()V"
consCall
returnType
stmt
ExplicitConsCall
Stmt
Expr
SingleTypeRef
Stmt
TypeRef
resolvedType = "void"
EmptyStmt
Stmt
index = 0

Comments



Node type	EXPLICIT CONSTRUCTOR CALL

Java code example

```java
package main.classes;
public class ExplicitConsCall {
    ExplicitConsCall() {
        this(1);
    }
    ExplicitConsCall(int i) {
    }
}
```

Corresponding syntax graph

Comments

Node type	**EXPRESSION**

Expr

Java code example

Corresponding syntax graph

Comments

- This node type is abstract.

Node type	EXTENDED STRING LITERAL

ExtendedStringLit

Java code example

```java
package main.classes;
public class ExtendedStringLit {
    String s = "a" + "b";
}
```

Corresponding syntax graph

CompUnitDecl
type
ClassDecl
Stmt
TypeDecl
name = "ExtendedStringLit"
method
field
AbsMethodDecl
ConsDecl
signature = "ExtendedStringLit()V"
consCall
ExplicitConsCall
Stmt
AbsVarDecl
FieldDecl
Stmt
index = 0
name = "s"
static = false
initialization
type
Expr
ExtendedStringLit
Lit
Stmt
StringLit
value = "ab"
Expr
SingleTypeRef
Stmt
TypeRef
resolvedType = "java.lang.String"

Comments

- See the super type on page 80 for additional information.

Node type	FALSE LITERAL

FalseLit
value:Boolean

Java code example

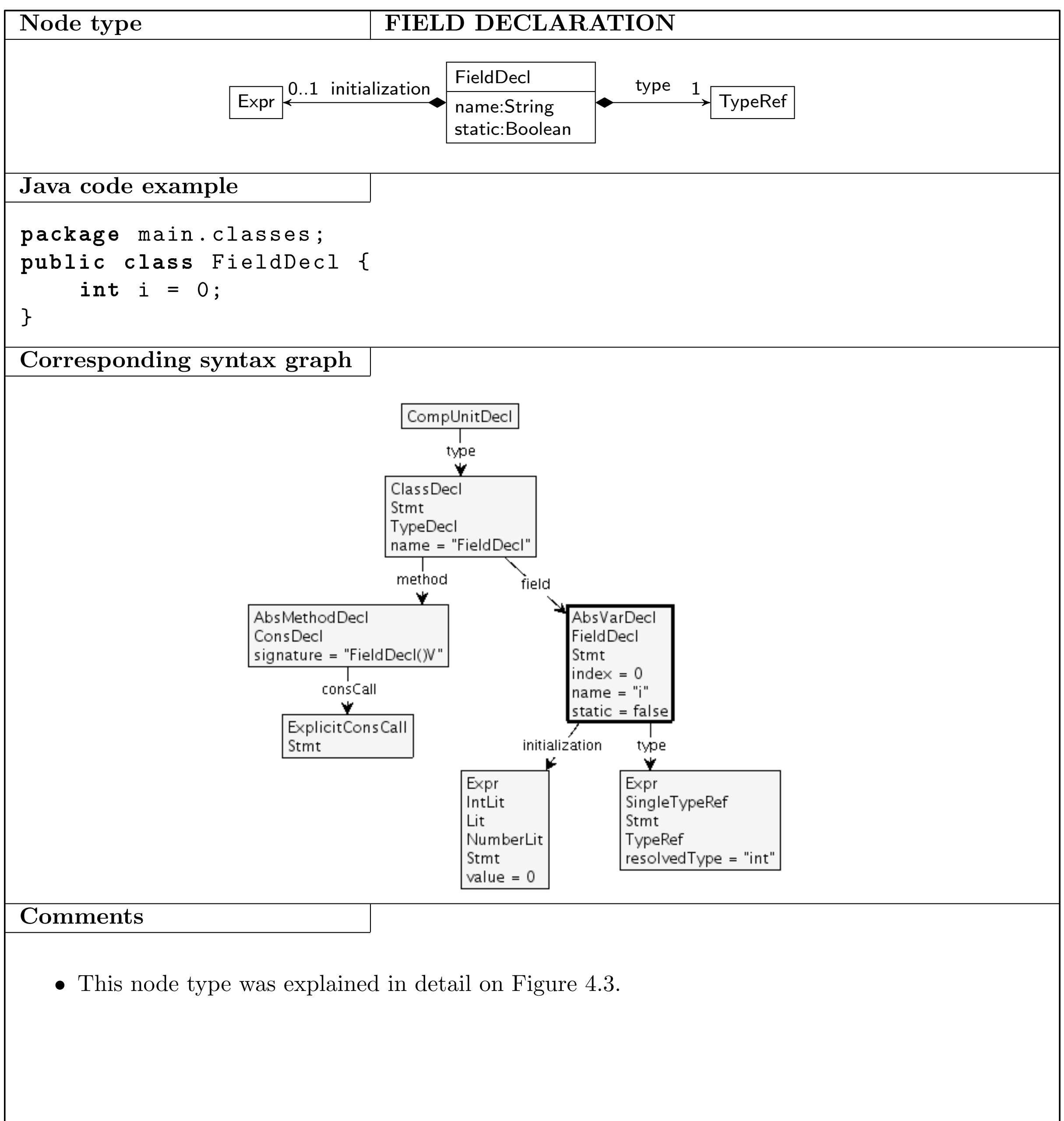
```
package main.classes;
public class FalseLit {
    boolean b = false;
}
```

Corresponding syntax graph

CompUnitDecl
type
ClassDecl
Stmt
TypeDecl
name = "FalseLit"
field
method
AbsVarDecl
FieldDecl
Stmt
index = 0
name = "b"
static = false
AbsMethodDecl
ConsDecl
signature = "FalseLit()V"
consCall
ExplicitConsCall
Stmt
type
initialization
Expr
SingleTypeRef
Stmt
TypeRef
resolvedType = "boolean"
Expr
FalseLit
Lit
MagicLit
Stmt
value = false

Comments

- The value attribute of this node type is always false.



Node type	FIELD DECLARATION

Java code example

```
package main.classes;
public class FieldDecl {
    int i = 0;
}
```

Corresponding syntax graph

Comments

- This node type was explained in detail on Figure 4.3.

Node type	FIELD REFERENCE

Java code example

```java
package main.classes;
public class FieldRef {
    int i = 0;
    public void method() {
        int j = this.i;
    }
}
```

Corresponding syntax graph

Comments

Node type	FLOAT LITERAL

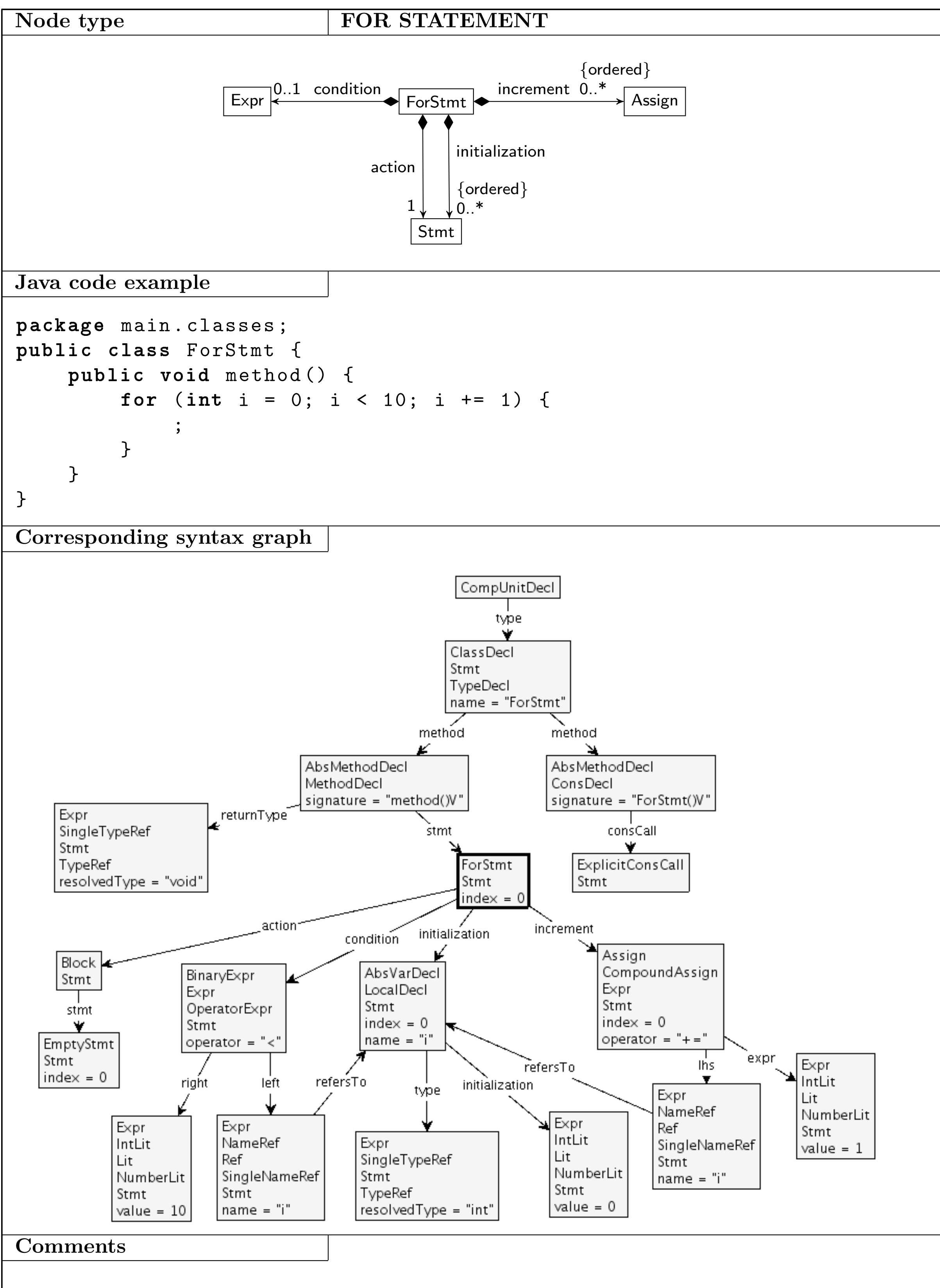
FloatLit
value:Float

Java code example

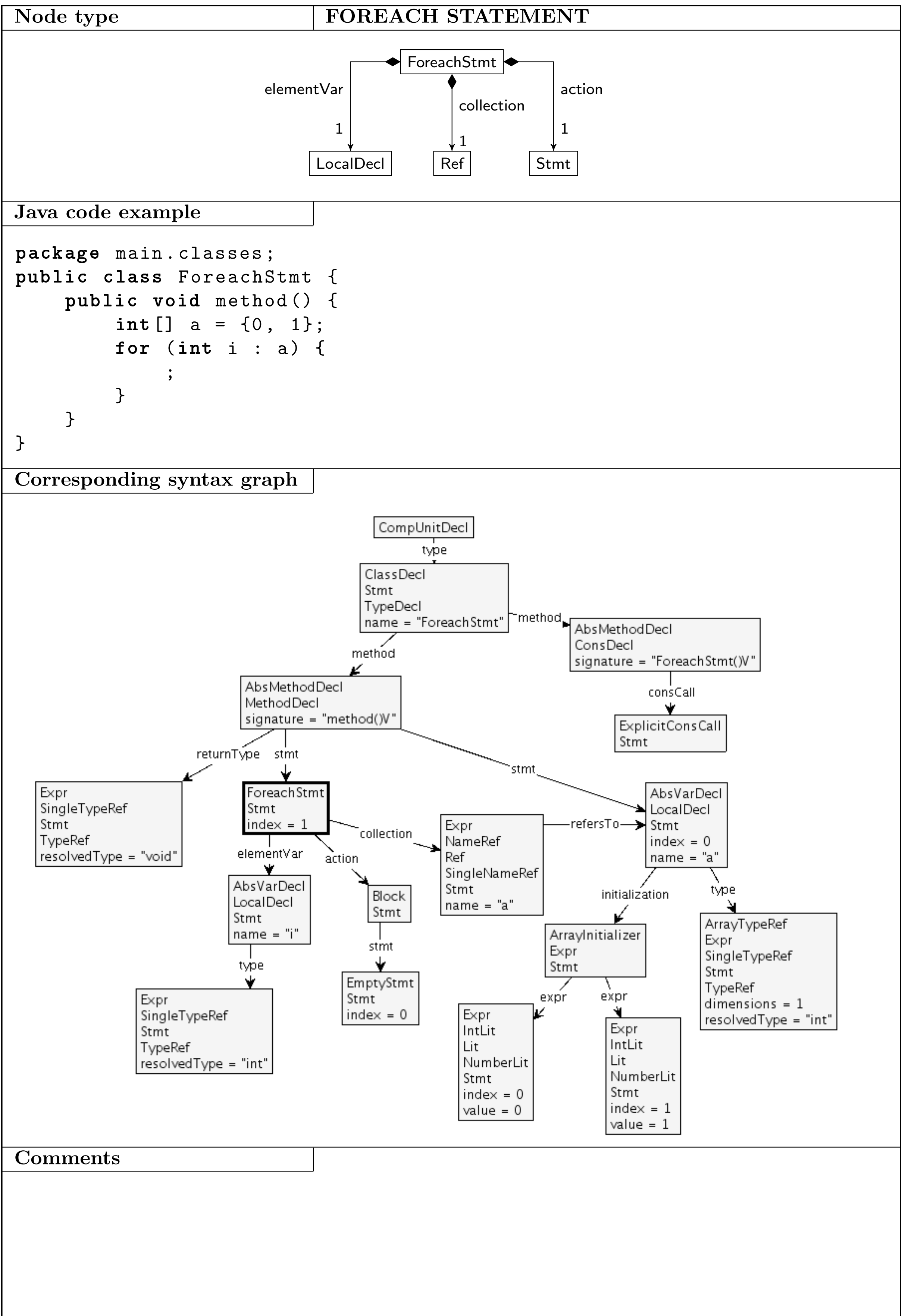
```
package main.classes;
public class FloatLit {
    float f = 1.0f;
}
```

Corresponding syntax graph

Comments



- This node type was explained in detail on Figure 4.5.



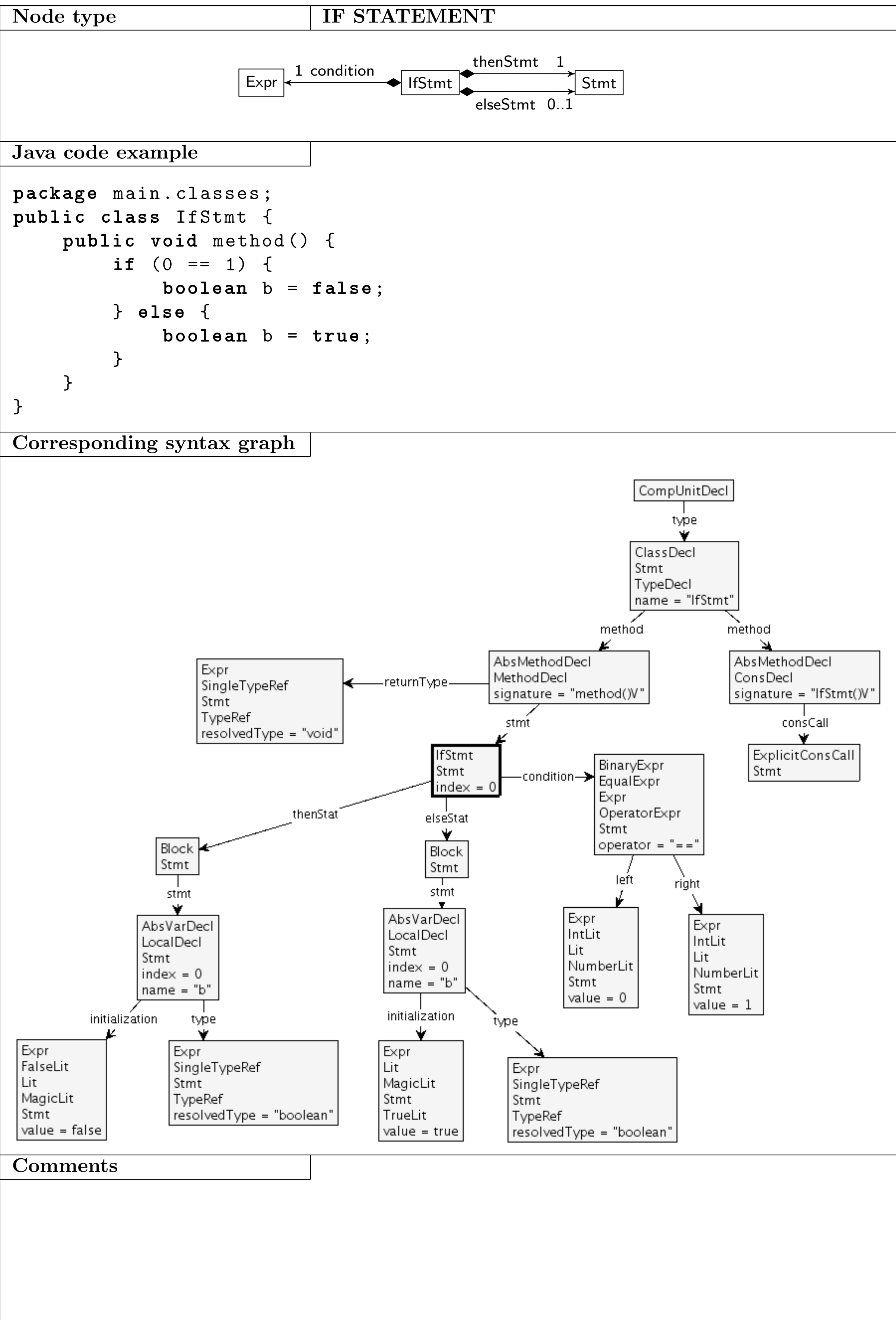
Node type	FOREACH STATEMENT

Java code example

```java
package main.classes;
public class ForeachStmt {
    public void method() {
        int[] a = {0, 1};
        for (int i : a) {
            ;
        }
    }
}
```

Corresponding syntax graph

Comments



Node type — **IF STATEMENT**

Java code example

```java
package main.classes;
public class IfStmt {
    public void method() {
        if (0 == 1) {
            boolean b = false;
        } else {
            boolean b = true;
        }
    }
}
```

Corresponding syntax graph

Comments

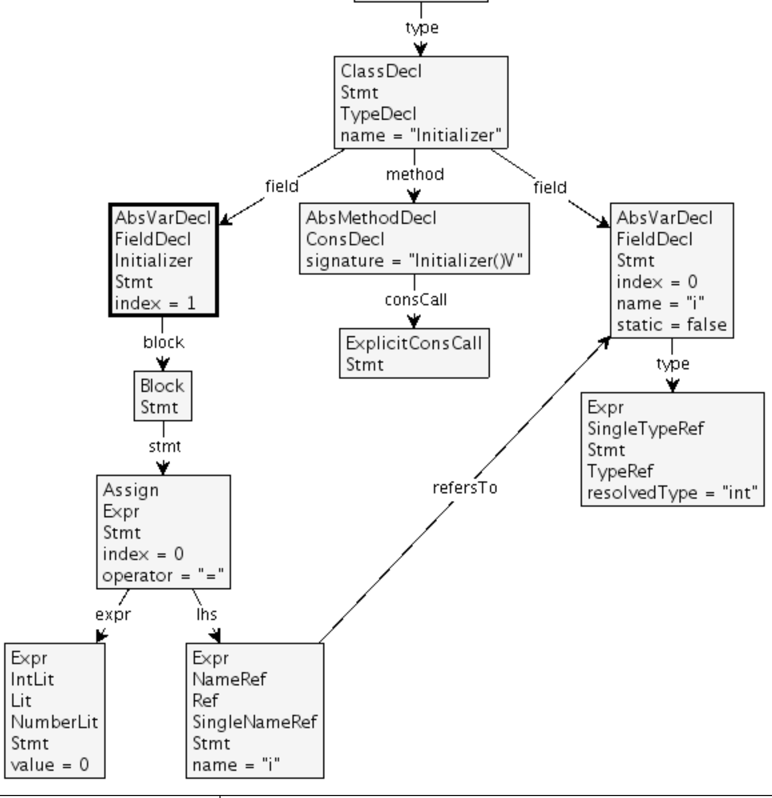
Node type	INITIALIZER

Initializer —block 1→ Block

Java code example

```java
package main.classes;
public class Initializer {
    int i;
    {
        i = 0;
    };
}
```

Corresponding syntax graph



Comments

- This node type represents the Java language elements called *Instance Initializers*.

Node type	INSTANCEOF EXPRESSION

Java code example

```
package main.classes;
public class InstanceOfExpr {
    boolean b = (null instanceof String);
}
```

Corresponding syntax graph

Comments

Node type	INT LITERAL

IntLit
value:Integer

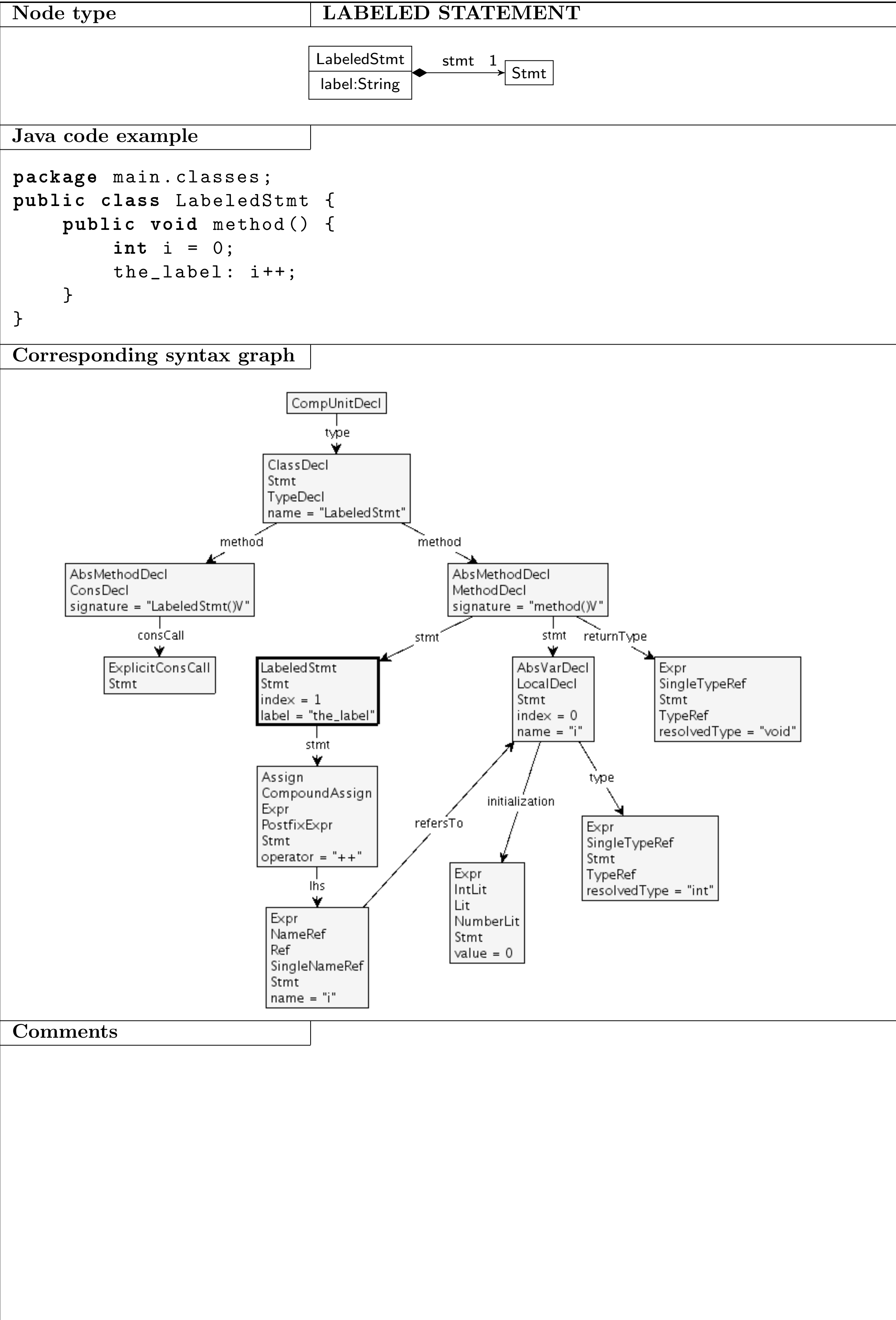
Java code example

```
package main.classes;
public class IntLit {
    int i = 42;
}
```

Corresponding syntax graph

CompUnitDecl
type
ClassDecl
Stmt
TypeDecl
name = "IntLit"
method
field
AbsMethodDecl
ConsDecl
signature = "IntLit()V"
consCall
ExplicitConsCall
Stmt
AbsVarDecl
FieldDecl
Stmt
index = 0
name = "i"
static = false
initialization
type
Expr
IntLit
Lit
NumberLit
Stmt
value = 42
Expr
SingleTypeRef
Stmt
TypeRef
resolvedType = "int"

Comments



Node type	LABELED STATEMENT

Java code example

```java
package main.classes;
public class LabeledStmt {
    public void method() {
        int i = 0;
        the_label: i++;
    }
}
```

Corresponding syntax graph

Comments

Node type	LITERAL

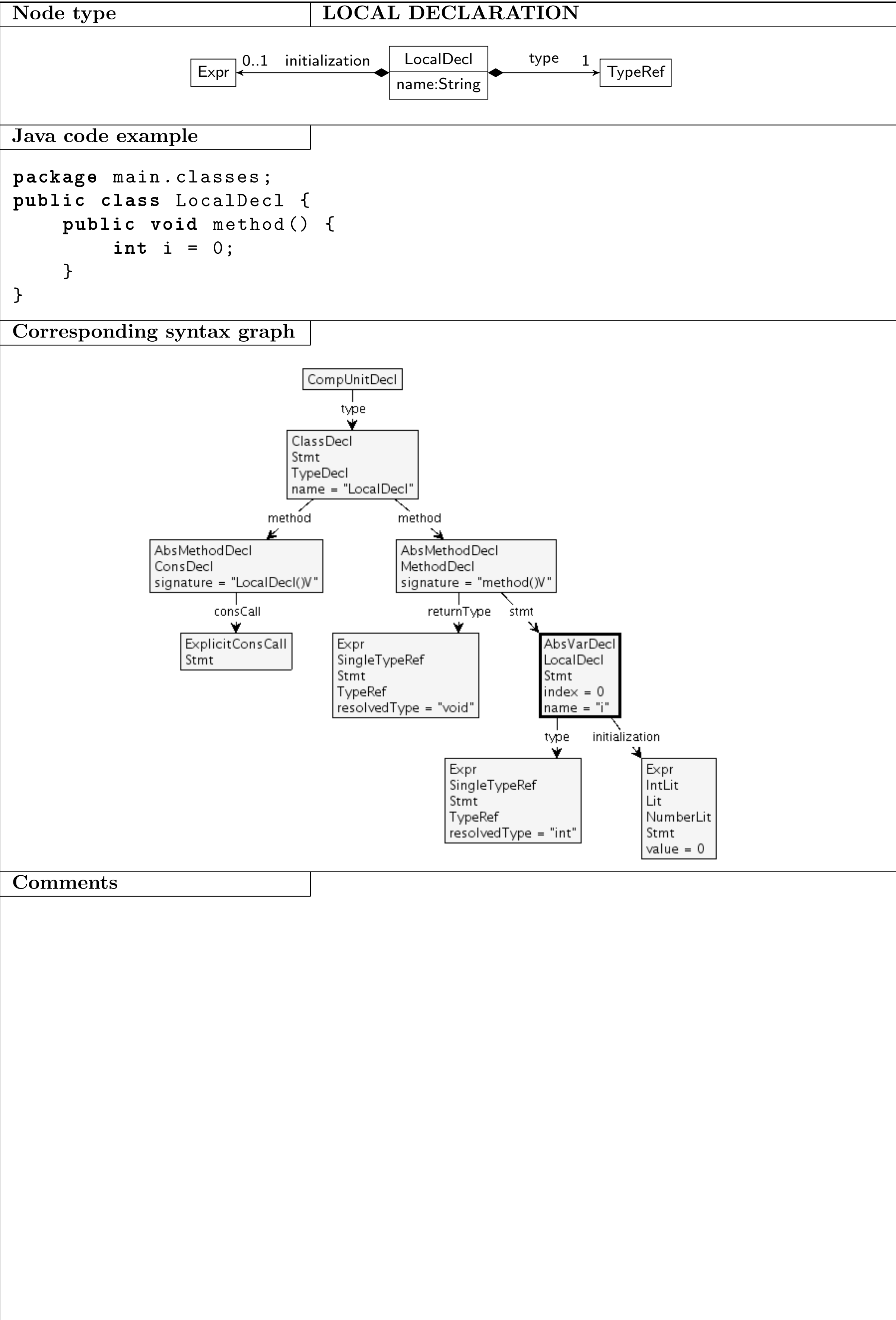
Lit

Java code example

Corresponding syntax graph

Comments

- This node type is abstract.



Node type	LOCAL DECLARATION

Java code example

```java
package main.classes;
public class LocalDecl {
    public void method() {
        int i = 0;
    }
}
```

Corresponding syntax graph

Comments

Node type	LONG LITERAL

LongLit
value:Long

Java code example

```
package main.classes;
public class LongLit {
    long l = 1L;
}
```

Corresponding syntax graph

CompUnitDecl
type →
ClassDecl
Stmt
TypeDecl
name = "LongLit"

method →
AbsMethodDecl
ConsDecl
signature = "LongLit()V"

consCall →
ExplicitConsCall
Stmt

field →
AbsVarDecl
FieldDecl
Stmt
index = 0
name = "l"
static = false

initialization →
Expr
Lit
LongLit
NumberLit
Stmt
value = 1

type →
Expr
SingleTypeRef
Stmt
TypeRef
resolvedType = "long"

Comments

Node type	**MAGIC LITERAL**

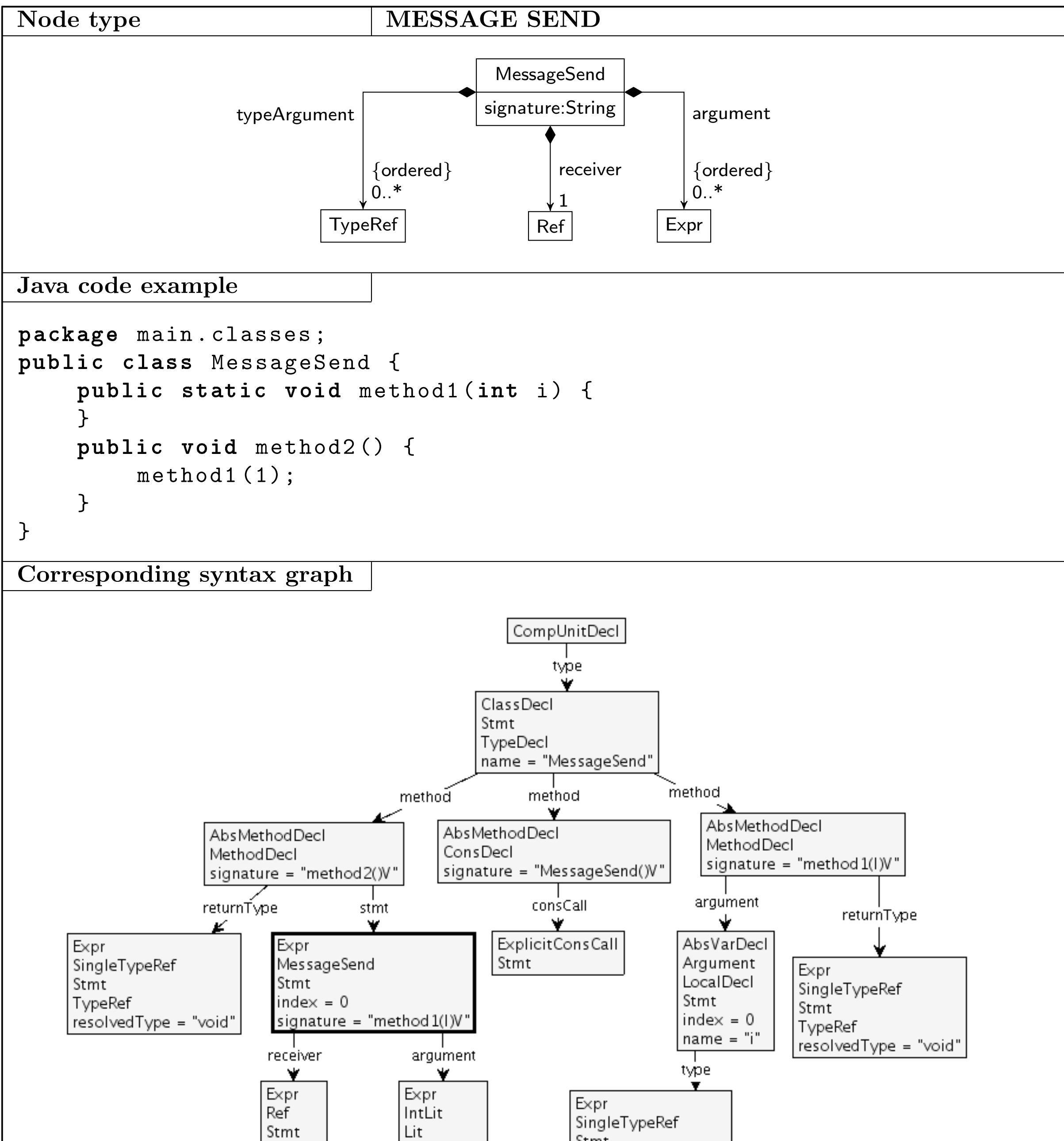
MagicLit

Java code example

Corresponding syntax graph

Comments

- This node type is abstract. See also pages 49, 67, and 87 for the concrete subtypes of this node type.



Node type — **MESSAGE SEND**

Java code example

```java
package main.classes;
public class MessageSend {
    public static void method1(int i) {
    }
    public void method2() {
        method1(1);
    }
}
```

Corresponding syntax graph

Comments

- This node type was explained in detail on Figure 4.6.

Node type	METHOD DECLARATION

MethodDecl —returnType 0..1→ TypeRef

Java code example

```java
package main.classes;
public class MethodDecl {
    public int method(int i) throws ArrayIndexOutOfBoundsException {
        int array[] = {0, 1};
        return array[i];
    }
}
```

Corresponding syntax graph

CompUnitDecl
type
ClassDecl
Stmt
TypeDecl
name = "MethodDecl"
method
AbsMethodDecl
ConsDecl
signature = "MethodDecl()V"
consCall
ExplicitConsCall
Stmt
method
AbsMethodDecl
MethodDecl
signature = "method(I)I"
returnType
Expr
SingleTypeRef
Stmt
TypeRef
resolvedType = "int"
thrownException
Expr
SingleTypeRef
Stmt
TypeRef
resolvedType = "java.lang.ArrayIndexOutOfBoundsException"
argument
AbsVarDecl
Argument
LocalDecl
Stmt
index = 0
name = "i"
type
Expr
SingleTypeRef
Stmt
TypeRef
resolvedType = "int"
refersTo
Expr
NameRef
Ref
SingleNameRef
Stmt
name = "i"
stmt
AbsVarDecl
LocalDecl
Stmt
index = 0
name = "array"
initialization
ArrayInitializer
Expr
Stmt
expr
Expr
IntLit
Lit
NumberLit
Stmt
index = 0
value = 0
expr
Expr
IntLit
Lit
NumberLit
Stmt
index = 1
value = 1
type
ArrayTypeRef
Expr
SingleTypeRef
Stmt
TypeRef
dimensions = 1
resolvedType = "int"
stmt
ReturnStmt
Stmt
index = 1
expr
ArrayRef
Expr
Ref
Stmt
position
receiver
Expr
NameRef
Ref
SingleNameRef
Stmt
name = "array"
refersTo

Comments

- See the super type on page 19 for additional information.

Node type	**NAME REFERENCE**

NameRef —refersTo 1→ AbsVarDecl

Java code example

Corresponding syntax graph

Comments

- This node type is abstract. See also pages 73, and 78 for the concrete subtypes of this node type.
- This node type was explained in detail on Figure 4.7.

Node type	NULL LITERAL

NullLit
value:String

Java code example

```java
package main.classes;
public class NullLit {
    Object o;
    boolean b = (o == null);
}
```

Corresponding syntax graph

CompUnitDecl
type
ClassDecl
Stmt
TypeDecl
name = "NullLit"
field
method
field
AbsVarDecl
FieldDecl
Stmt
index = 1
name = "b"
static = false
AbsMethodDecl
ConsDecl
signature = "NullLit()V"
AbsVarDecl
FieldDecl
Stmt
index = 0
name = "o"
static = false
initialization
consCall
type
BinaryExpr
EqualExpr
Expr
OperatorExpr
Stmt
operator = "=="
ExplicitConsCall
Stmt
Expr
SingleTypeRef
Stmt
TypeRef
resolvedType = "boolean"
refersTo
type
right
left
Expr
Lit
MagicLit
NullLit
Stmt
value = "null"
Expr
NameRef
Ref
SingleNameRef
Stmt
name = "o"
Expr
SingleTypeRef
Stmt
TypeRef
resolvedType = "java.lang.Object"

Comments

- The value attribute of this node type is always the string "null".

Node type	NUMBER LITERAL

NumberLit

Java code example

Corresponding syntax graph

Comments

- This node type is abstract. See also pages 35, 44, 52, 58, and 62 for the concrete subtypes of this node type.

Node type	**OPERATOR EXPRESSION**

OperatorExpr

Java code example

Corresponding syntax graph

Comments

- This node type is abstract. See also pages 29, 40, 57, and 92 for the concrete subtypes of this node type.

Node type	POSTFIX EXPRESSION

PostfixExpr

Java code example

```
package main.classes;
public class PostfixExpr {
    int i = 0;
    int j = i++;
}
```

Corresponding syntax graph

CompUnitDecl

type

ClassDecl
Stmt
TypeDecl
name = "PostfixExpr"

method

field

field

AbsMethodDecl
ConsDecl
signature = "PostfixExpr()V"

consCall

ExplicitConsCall
Stmt

AbsVarDecl
FieldDecl
Stmt
index = 1
name = "j"
static = false

type

Expr
SingleTypeRef
Stmt
TypeRef
resolvedType = "int"

initialization

Assign
CompoundAssign
Expr
PostfixExpr
Stmt
operator = "++"

lhs

Expr
NameRef
Ref
SingleNameRef
Stmt
name = "i"

refersTo

AbsVarDecl
FieldDecl
Stmt
index = 0
name = "i"
static = false

type

Expr
SingleTypeRef
Stmt
TypeRef
resolvedType = "int"

initialization

Expr
IntLit
Lit
NumberLit
Stmt
value = 0

Comments

- See the super types on page 27 and 39 for additional information.

Node type	PREFIX EXPRESSION

PrefixExpr

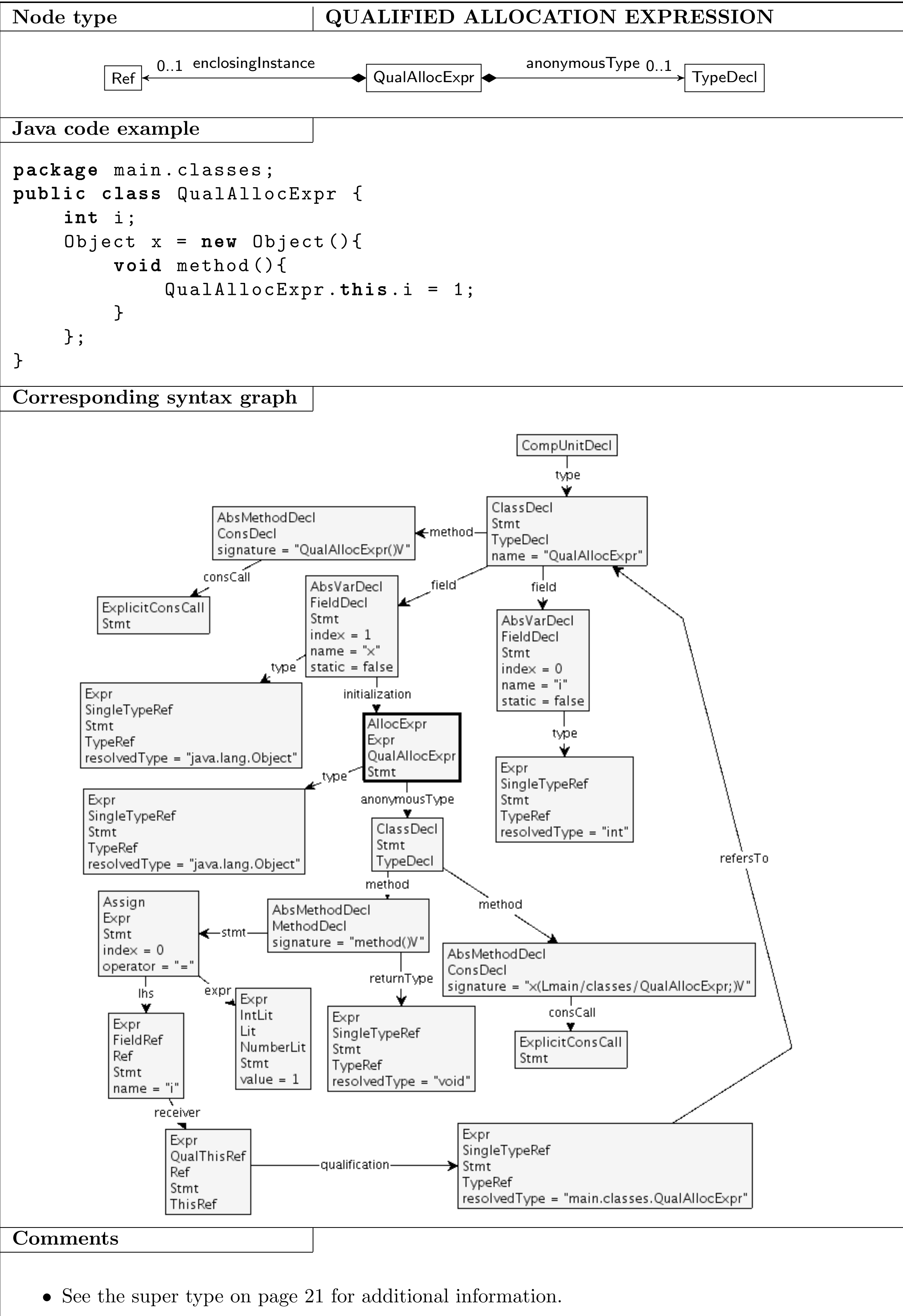
Java code example

```java
package main.classes;
public class PrefixExpr {
    int i = 0;
    int j = ++i;
}
```

Corresponding syntax graph

Comments

- See the super types on page 27 and 39 for additional information.



Node type	QUALIFIED ALLOCATION EXPRESSION

Java code example

```java
package main.classes;
public class QualAllocExpr {
    int i;
    Object x = new Object(){
        void method(){
            QualAllocExpr.this.i = 1;
        }
    };
}
```

Corresponding syntax graph

Comments

- See the super type on page 21 for additional information.

Node type	QUALIFIED NAME REFERENCE

QualNameRef

Java code example

```java
package main.classes;
import java.io.OutputStream;
public class QualNameRef {
    OutputStream out = System.out;
}
```

Corresponding syntax graph

CompUnitDecl

type

ClassDecl
Stmt
TypeDecl
name = "QualNameRef"

method

field

AbsMethodDecl
ConsDecl
signature = "QualNameRef()V"

consCall

ExplicitConsCall
Stmt

AbsVarDecl
FieldDecl
Stmt
index = 0
name = "out"
static = false

initialization

type

Expr
NameRef
QualNameRef
Ref
Stmt
name = "System.out"

Expr
SingleTypeRef
Stmt
TypeRef
resolvedType = "java.io.OutputStream"

Comments

- See the super type on page 66 for additional information.

Node type	QUALIFIED SUPER REFERENCE

QualSuperRef

Java code example

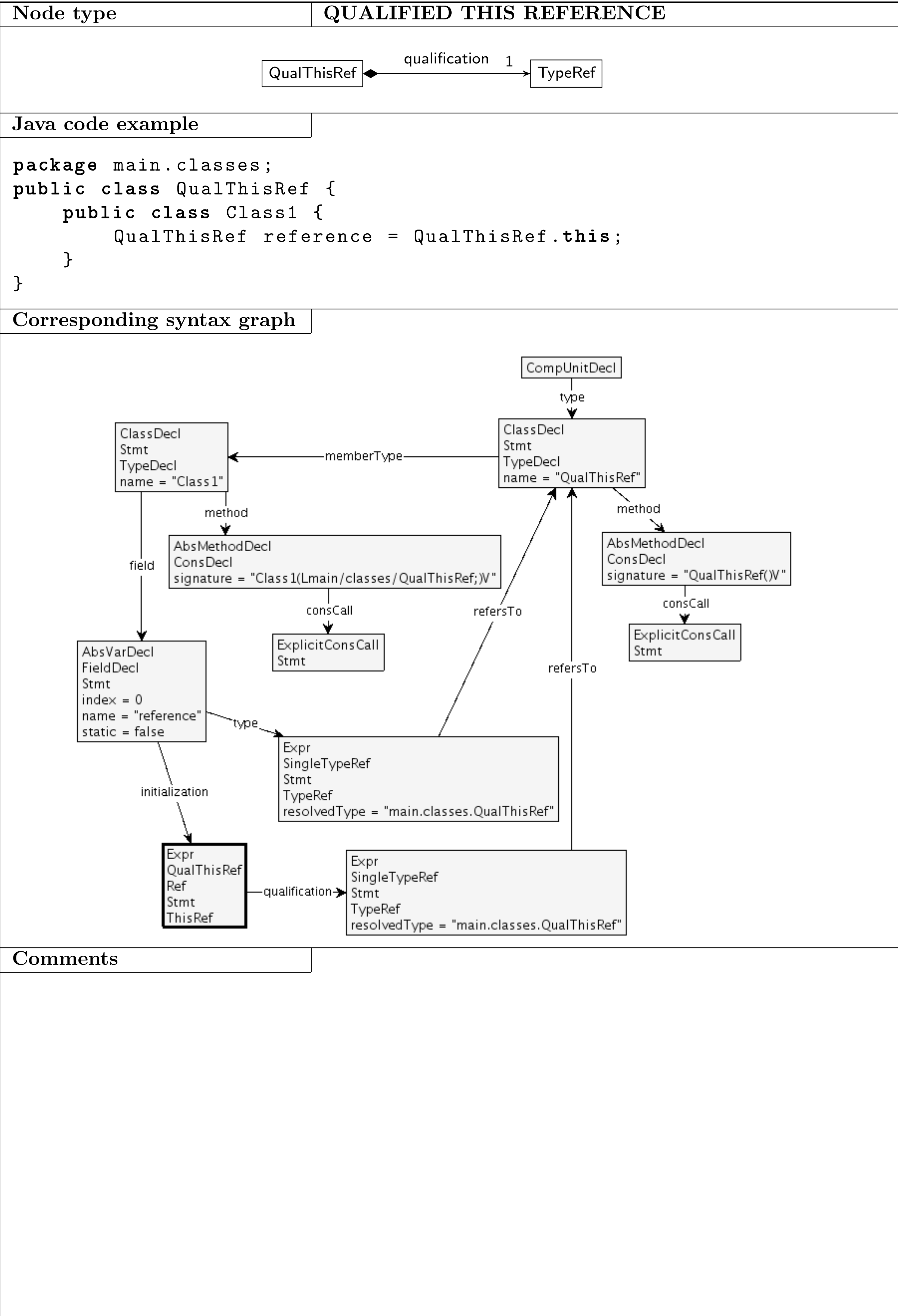
```
package main.classes;
public class QualSuperRef {
    Object x = QualSuperRef.super;
}
```

Corresponding syntax graph

CompUnitDecl
type
ClassDecl
Stmt
TypeDecl
name = "QualSuperRef"
method
field
AbsVarDecl
FieldDecl
Stmt
index = 0
name = "x"
static = false
AbsMethodDecl
ConsDecl
signature = "QualSuperRef()V"
consCall
ExplicitConsCall
Stmt
refersTo
initialization
type
Expr
QualSuperRef
QualThisRef
Ref
Stmt
ThisRef
Expr
SingleTypeRef
Stmt
TypeRef
resolvedType = "java.lang.Object"
qualification
Expr
SingleTypeRef
Stmt
TypeRef
resolvedType = "main.classes.QualSuperRef"

Comments

- See the super type on page 75 for additional information.



Node type	QUALIFIED THIS REFERENCE

Java code example

```
package main.classes;
public class QualThisRef {
    public class Class1 {
        QualThisRef reference = QualThisRef.this;
    }
}
```

Corresponding syntax graph

Comments

Node type	**REFERENCE**

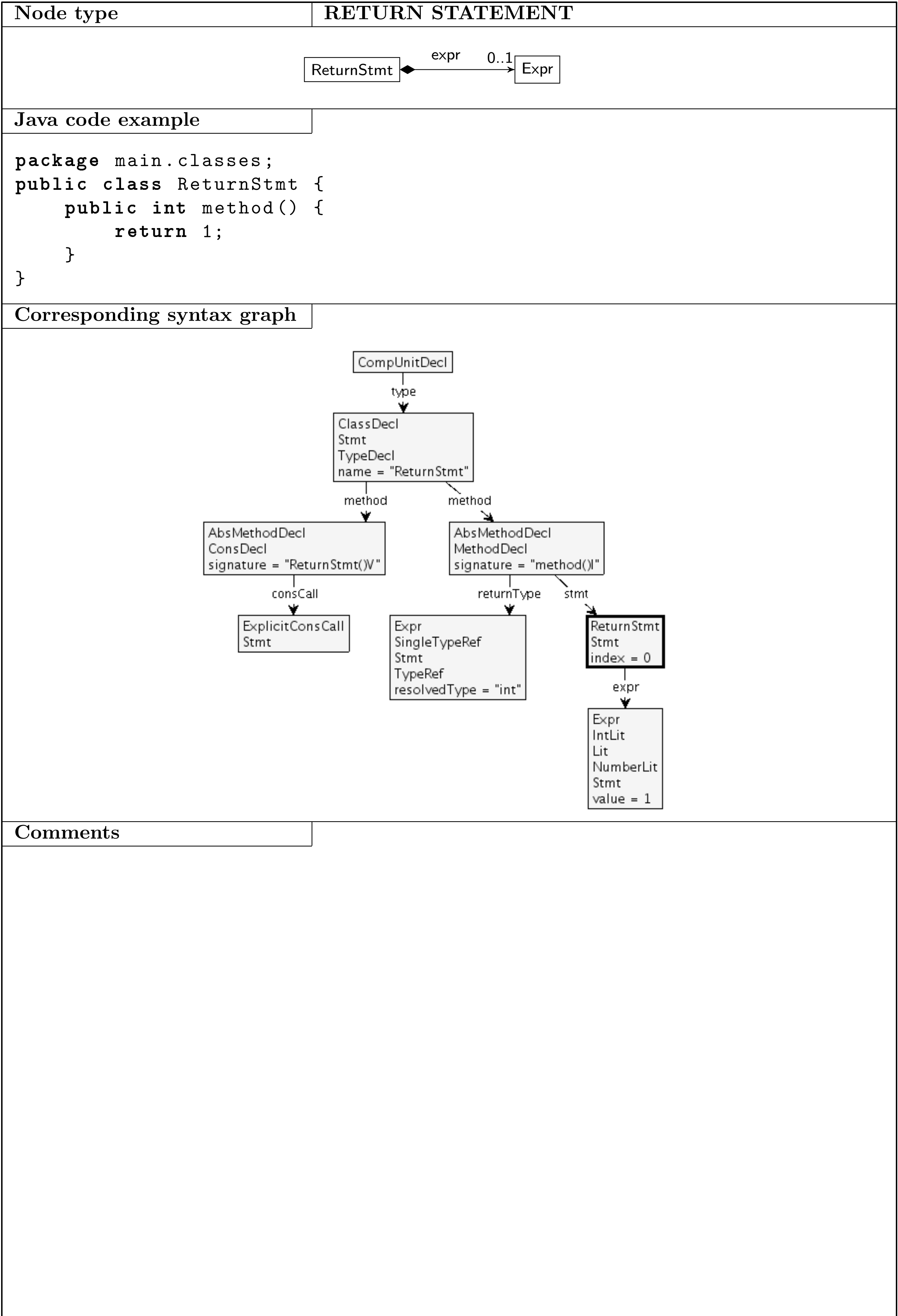
Ref

Java code example

Corresponding syntax graph

Comments

- This node type is abstract. See also pages 25, 51, 66, and 85 for the direct subtypes of this node type.



Node type	RETURN STATEMENT

Java code example

```java
package main.classes;
public class ReturnStmt {
    public int method() {
        return 1;
    }
}
```

Corresponding syntax graph

Comments

Node type	SINGLE NAME REFERENCE

SingleNameRef

Java code example

```
package main.classes;
public class SingleNameRef {
    int i = 0;
    int j = i;
}
```

Corresponding syntax graph

CompUnitDecl
type
ClassDecl
Stmt
TypeDecl
name = "SingleNameRef"
field
field
method
AbsVarDecl
FieldDecl
Stmt
index = 1
name = "j"
static = false
AbsVarDecl
FieldDecl
Stmt
index = 0
name = "i"
static = false
AbsMethodDecl
ConsDecl
signature = "SingleNameRef()V"
consCall
ExplicitConsCall
Stmt
type
initialization
refersTo
type
initialization
Expr
SingleTypeRef
Stmt
TypeRef
resolvedType = "int"
Expr
NameRef
Ref
SingleNameRef
Stmt
name = "i"
Expr
SingleTypeRef
Stmt
TypeRef
resolvedType = "int"
Expr
IntLit
Lit
NumberLit
Stmt
value = 0

Comments

- See the super type on page 66 for additional information.

Node type	**STATEMENT**

Stmt

Java code example

Corresponding syntax graph

Comments

- This node type is abstract.

Node type	STRING LITERAL

StringLit
value:String

Java code example

```java
package main.classes;
public class StringLit {
    String s = "abc";
}
```

Corresponding syntax graph

CompUnitDecl
type
ClassDecl
Stmt
TypeDecl
name = "StringLit"
field
method
AbsVarDecl
FieldDecl
Stmt
index = 0
name = "s"
static = false
AbsMethodDecl
ConsDecl
signature = "StringLit()V"
consCall
ExplicitConsCall
Stmt
type
initialization
Expr
SingleTypeRef
Stmt
TypeRef
resolvedType = "java.lang.String"
Expr
Lit
Stmt
StringLit
value = "abc"

Comments

Node type	SUBROUTINE STATEMENT

SubRoutineStmt

Java code example

Corresponding syntax graph

Comments

- This node type is abstract. See also pages 84, and 88 for the concrete subtypes of this node type.

Node type	SUPER REFERENCE

SuperRef

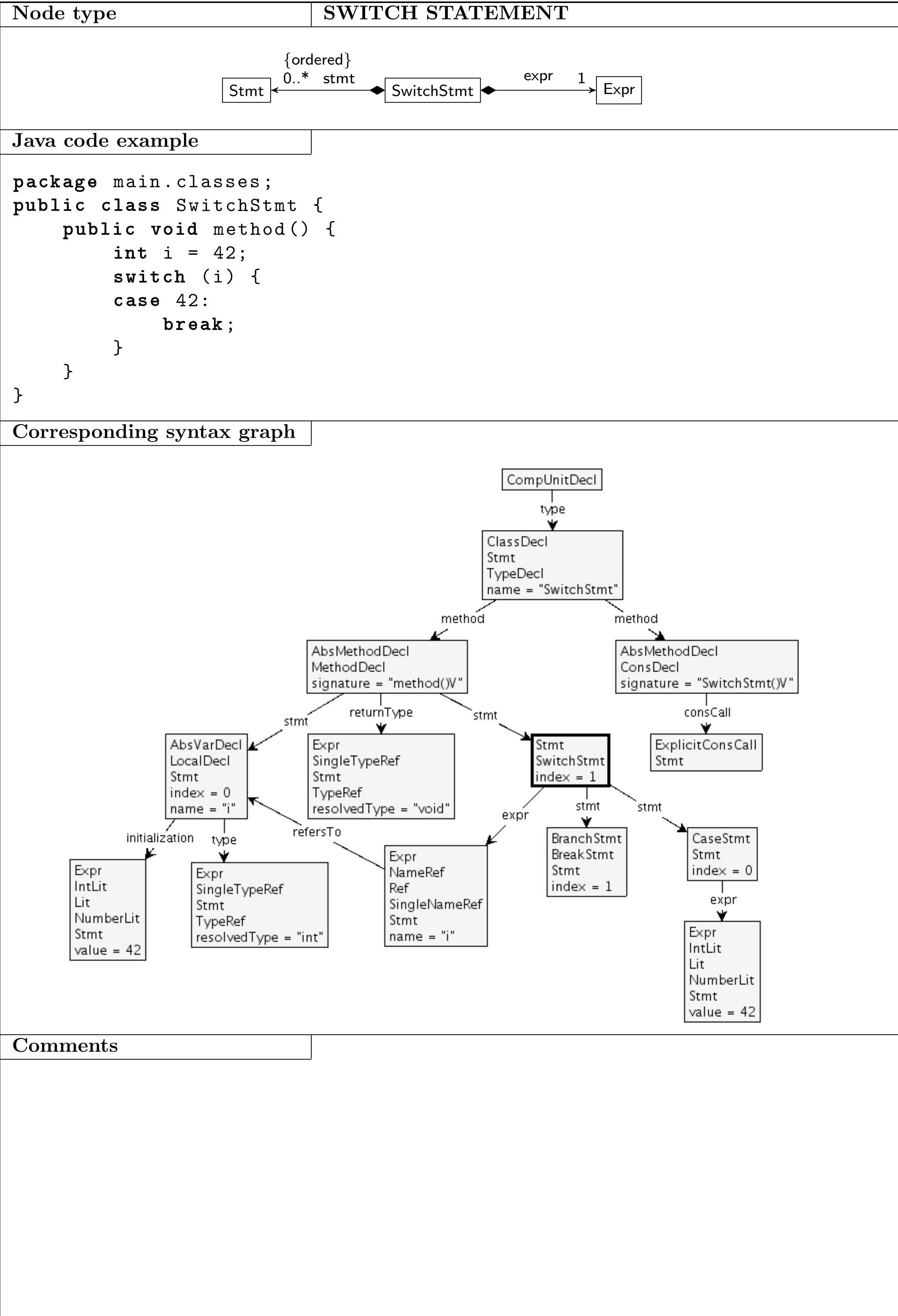
Java code example

```java
package main.classes;
public class SuperRef {
    public void method() {
        super.notify();
    }
}
```

Corresponding syntax graph

CompUnitDecl
type
ClassDecl
Stmt
TypeDecl
name = "SuperRef"
method
method
AbsMethodDecl
ConsDecl
signature = "SuperRef()V"
AbsMethodDecl
MethodDecl
signature = "method()V"
consCall
returnType
stmt
ExplicitConsCall
Stmt
Expr
SingleTypeRef
Stmt
TypeRef
resolvedType = "void"
Expr
MessageSend
Stmt
index = 0
signature = "notify()V"
receiver
Expr
Ref
Stmt
SuperRef
ThisRef

Comments



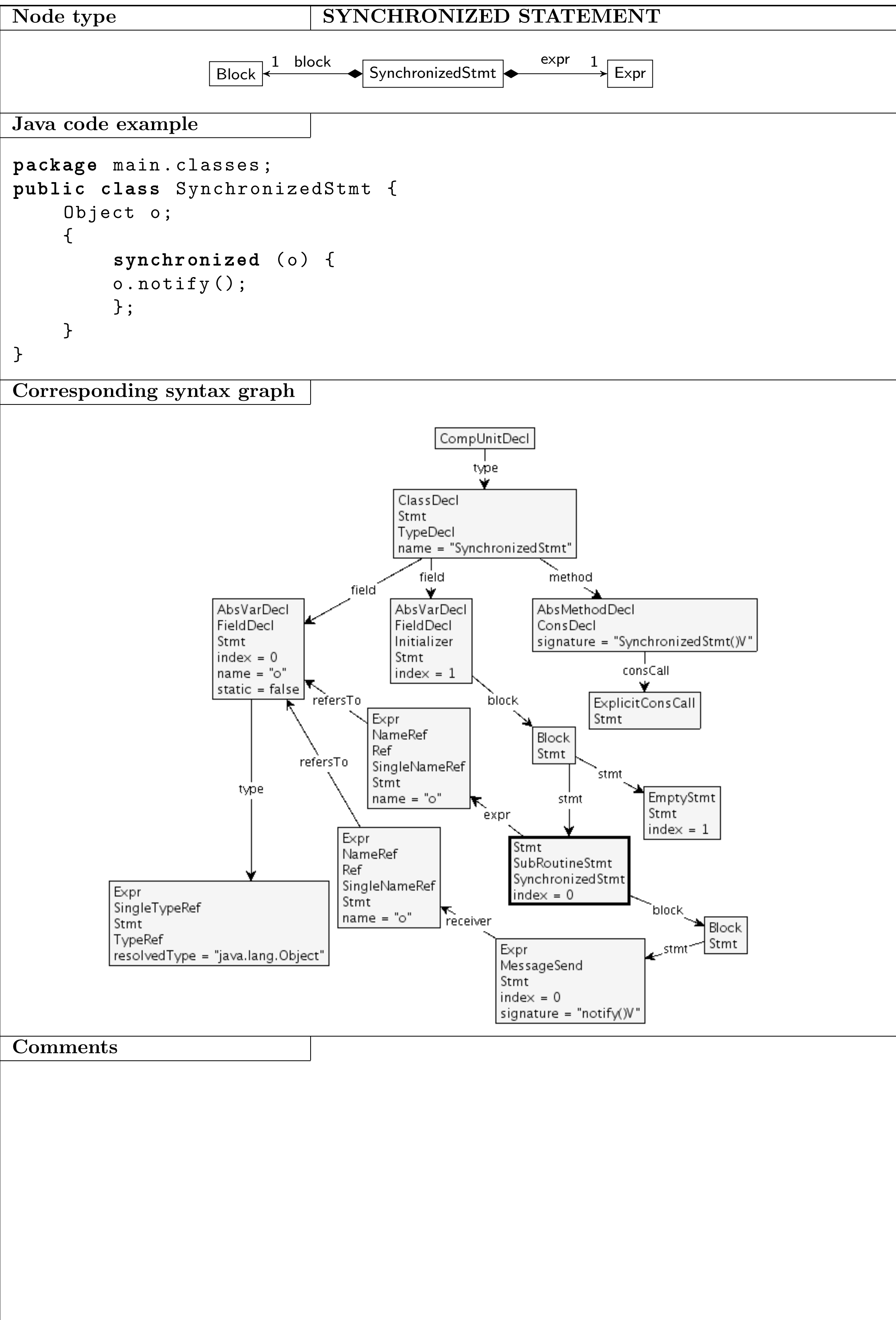
Node type	SWITCH STATEMENT

Java code example

```
package main.classes;
public class SwitchStmt {
    public void method() {
        int i = 42;
        switch (i) {
        case 42:
            break;
        }
    }
}
```

Corresponding syntax graph

Comments



Node type	SYNCHRONIZED STATEMENT

Java code example

```
package main.classes;
public class SynchronizedStmt {
    Object o;
    {
        synchronized (o) {
        o.notify();
        };
    }
}
```

Corresponding syntax graph

Comments

Node type | **THIS REFERENCE**

ThisRef

Java code example

```
package main.classes;
public class ThisRef {
    public Object method() {
        return this;
    }
}
```

Corresponding syntax graph

CompUnitDecl
type
ClassDecl
Stmt
TypeDecl
name = "ThisRef"
method
method
AbsMethodDecl
ConsDecl
signature = "ThisRef()V"
AbsMethodDecl
MethodDecl
signature = "method()Ljava/lang/Object;"
consCall
stmt
returnType
ExplicitConsCall
Stmt
ReturnStmt
Stmt
index = 0
Expr
SingleTypeRef
Stmt
TypeRef
resolvedType = "java.lang.Object"
expr
Expr
Ref
Stmt
ThisRef

Comments

Node type	THROW STATEMENT

ThrowStmt —exception 1→ Expr

Java code example

```
package main.classes;
public class ThrowStmt {
    public void method() {
        throw new NullPointerException();
    }
}
```

Corresponding syntax graph

CompUnitDecl
—type→ ClassDecl / Stmt / TypeDecl / name = "ThrowStmt"
—method→ AbsMethodDecl / ConsDecl / signature = "ThrowStmt()V"
—consCall→ ExplicitConsCall / Stmt
—method→ AbsMethodDecl / MethodDecl / signature = "method()V"
—stmt→ Stmt / ThrowStmt / index = 0
—exception→ AllocExpr / Expr / Stmt
—type→ Expr / SingleTypeRef / Stmt / TypeRef / resolvedType = "java.lang.NullPointerException"
—returnType→ Expr / SingleTypeRef / Stmt / TypeRef / resolvedType = "void"

Comments

Node type	TRUE LITERAL

TrueLit
value:Boolean

Java code example

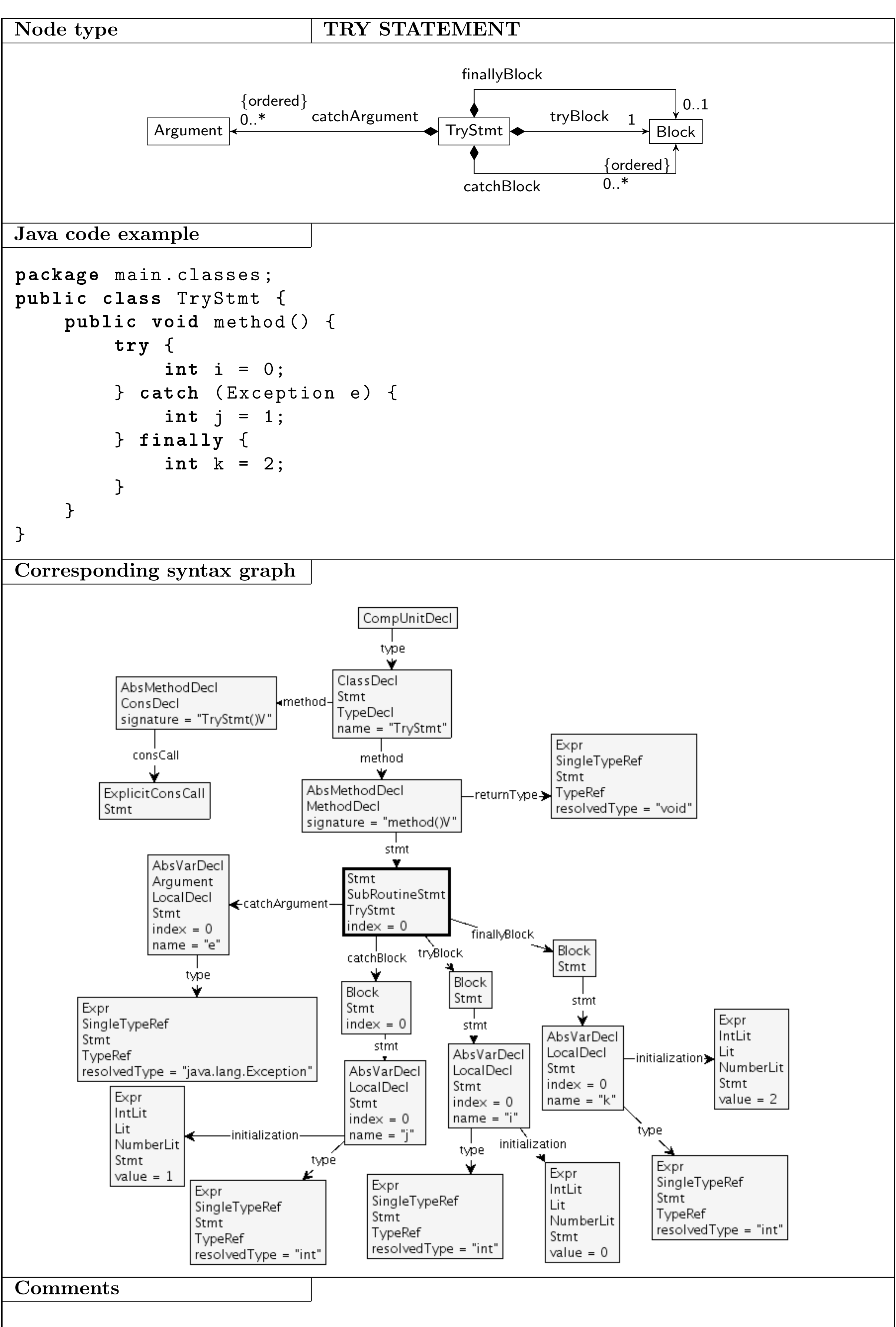
```
package main.classes;
public class TrueLit {
    boolean b = true;
}
```

Corresponding syntax graph

CompUnitDecl
type
ClassDecl
Stmt
TypeDecl
name = "TrueLit"
method
field
AbsMethodDecl
ConsDecl
signature = "TrueLit()V"
consCall
ExplicitConsCall
Stmt
AbsVarDecl
FieldDecl
Stmt
index = 0
name = "b"
static = false
initialization
type
Expr
Lit
MagicLit
Stmt
TrueLit
value = true
Expr
SingleTypeRef
Stmt
TypeRef
resolvedType = "boolean"

Comments

- The value attribute of this node type is always true.



Node type	TRY STATEMENT

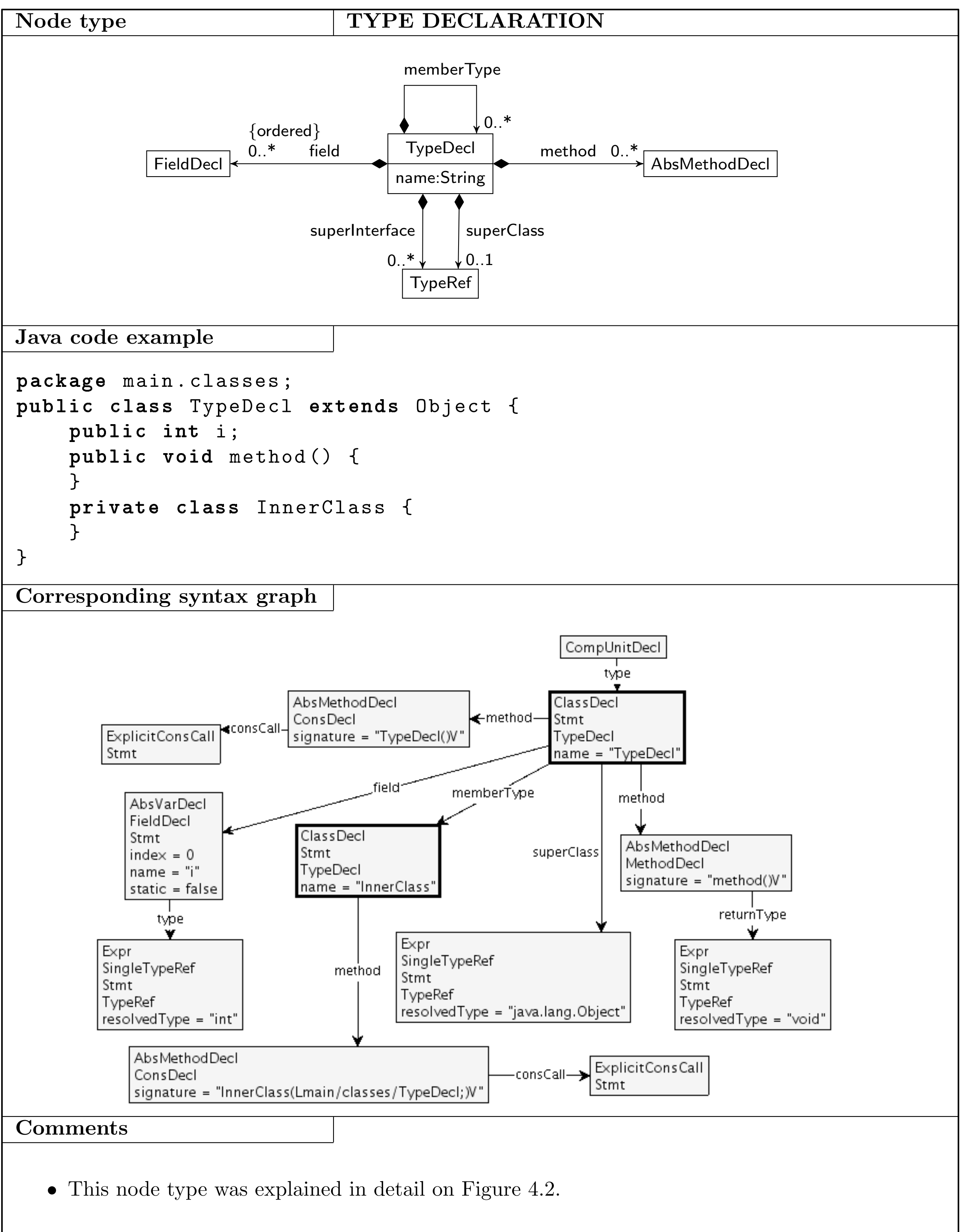
Java code example

```java
package main.classes;
public class TryStmt {
    public void method() {
        try {
            int i = 0;
        } catch (Exception e) {
            int j = 1;
        } finally {
            int k = 2;
        }
    }
}
```

Corresponding syntax graph

Comments

Node type — **TYPE DECLARATION**



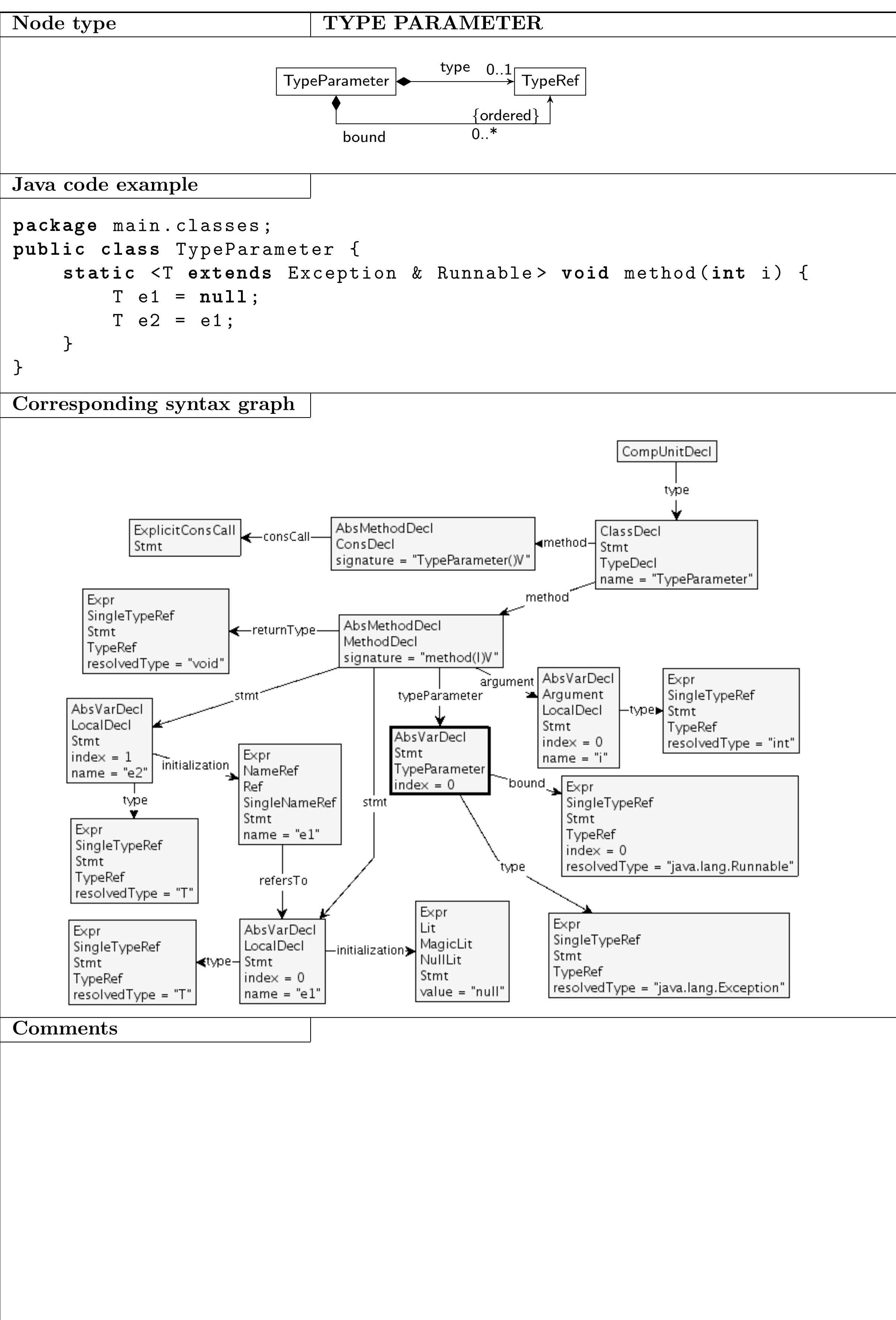
Java code example

```java
package main.classes;
public class TypeDecl extends Object {
    public int i;
    public void method() {
    }
    private class InnerClass {
    }
}
```

Corresponding syntax graph

Comments

- This node type was explained in detail on Figure 4.2.



Node type	TYPE PARAMETER

Java code example

```java
package main.classes;
public class TypeParameter {
    static <T extends Exception & Runnable> void method(int i) {
        T e1 = null;
        T e2 = e1;
    }
}
```

Corresponding syntax graph

Comments

Node type	TYPE REFERENCE

TypeRef
refersTo 1
TypeDecl

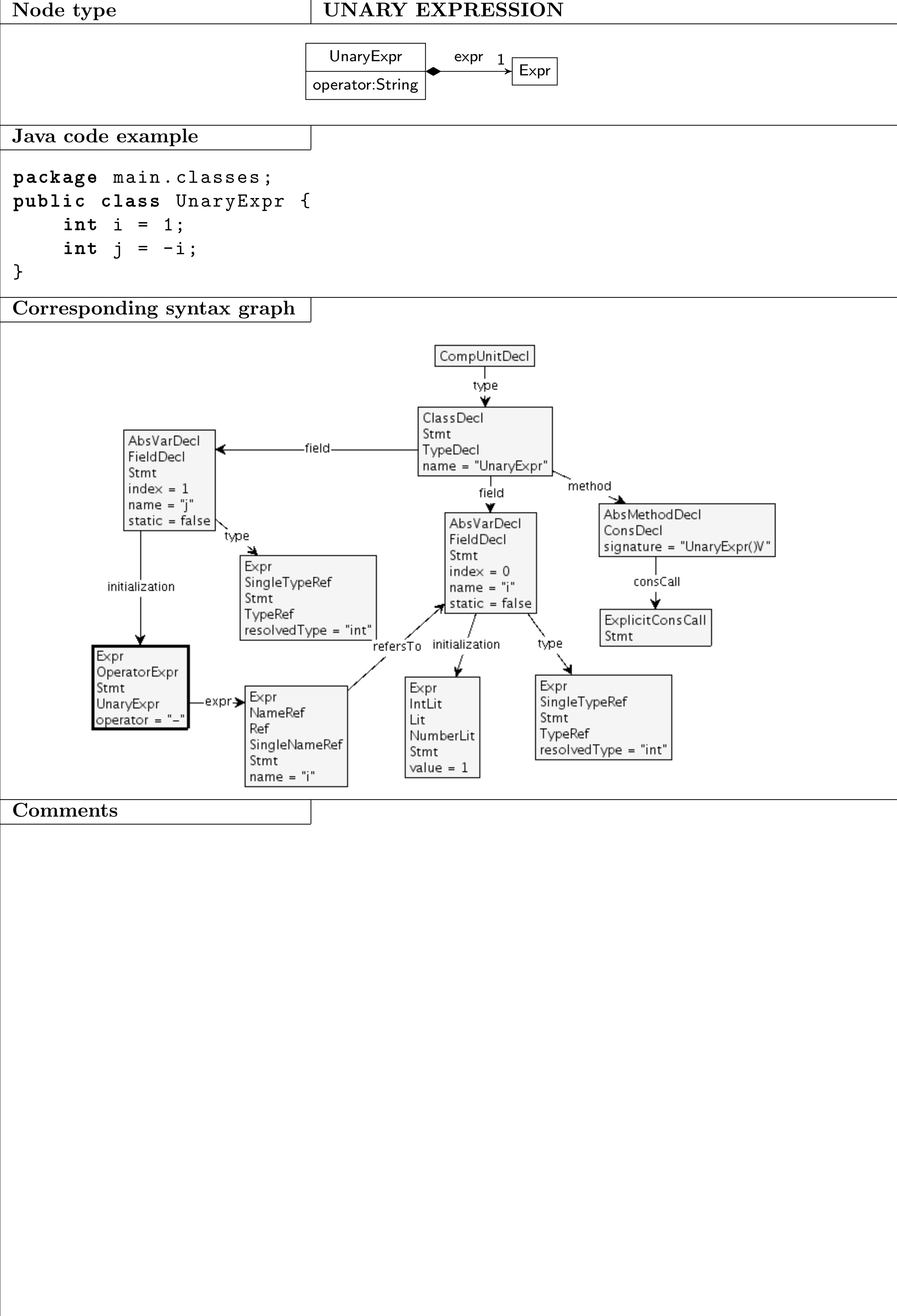
Java code example

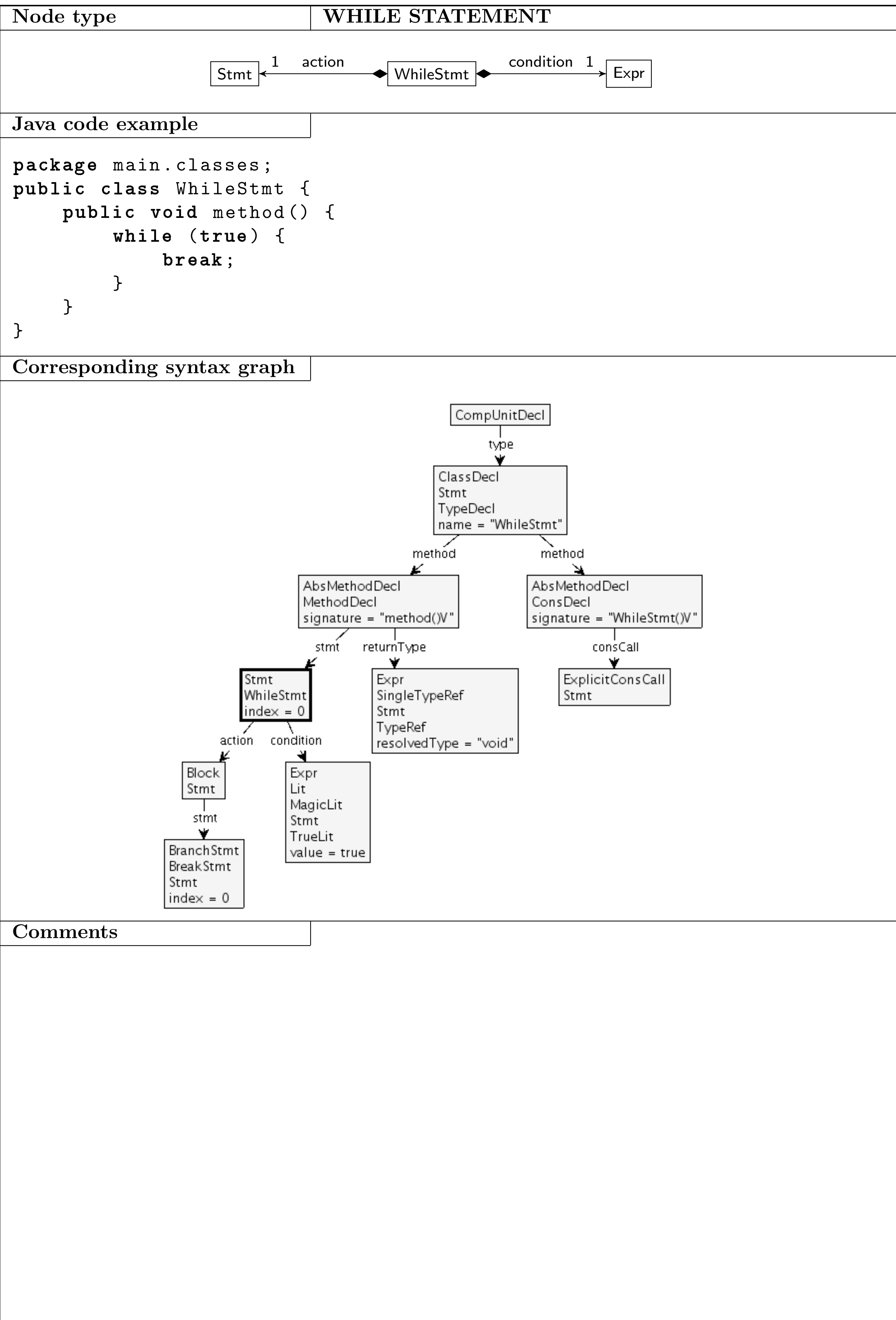
```
package main.classes;
public class TypeRef {
    TypeRef ref;
}
```

Corresponding syntax graph

CompUnitDecl
type
ClassDecl
Stmt
TypeDecl
name = "TypeRef"
method
field
refersTo
AbsMethodDecl
ConsDecl
signature = "TypeRef()V"
consCall
ExplicitConsCall
Stmt
AbsVarDecl
FieldDecl
Stmt
index = 0
name = "ref"
static = false
type
Expr
SingleTypeRef
Stmt
TypeRef
resolvedType = "main.classes.TypeRef"

Comments


Node type
UNARY EXPRESSION
UnaryExpr
operator:String
expr
1
Expr
Java code example
package main.classes;
public class UnaryExpr {
int i = 1;
int j = -i;
}
Corresponding syntax graph
CompUnitDecl
type
ClassDecl
Stmt
TypeDecl
name = "UnaryExpr"
field
AbsVarDecl
FieldDecl
Stmt
index = 1
name = "j"
static = false
type
Expr
SingleTypeRef
Stmt
TypeRef
resolvedType = "int"
initialization
Expr
OperatorExpr
Stmt
UnaryExpr
operator = "–"
expr
Expr
NameRef
Ref
SingleNameRef
Stmt
name = "i"
refersTo
field
AbsVarDecl
FieldDecl
Stmt
index = 0
name = "i"
static = false
initialization
Expr
IntLit
Lit
NumberLit
Stmt
value = 1
type
Expr
SingleTypeRef
Stmt
TypeRef
resolvedType = "int"
method
AbsMethodDecl
ConsDecl
signature = "UnaryExpr()V"
consCall
ExplicitConsCall
Stmt
Comments



Node type	WHILE STATEMENT

Java code example

```java
package main.classes;
public class WhileStmt {
    public void method() {
        while (true) {
            break;
        }
    }
}
```

Corresponding syntax graph

Comments