Jumbala Action Language Specification
Version 1.20

Jori Dubrovin

12th June 2007

Contents

Chapter 1

Introduction

The Jumbala language has been designed to function as an action language in behavioral UML models. The idea is that action specifications in UML state machines are written in Jumbala by the user. The purpose of this document is to describe the details of Jumbala as a stand-alone programming language. An overview of the language and the design decisions behind it and a description of the connection between Jumbala and UML is given in [1].

Jumbala is based on version 5.0 of the Java programming language. Correspondingly, this document is based on the Java Language Specification [2]. Jumbala is almost a simplified version of Java, with only minor additions. For the most part, Jumbala follows the syntax and semantics of Java except for the differences resulting from the omission of certain features. We have tried to point out clearly the few deviations from the Java standard. Many of the intricacies that follow from the rules of Java are also present in Jumbala, and not all of them have been explicitly brought out in the following chapters. When in doubt, the reader is invited to consult the appropriate sections of the Java Specification for code examples that illustrate the subtleties. All the identified differences between Jumbala and Java have been listed in Appendix A.

1.1 The Syntactic Grammar

We use a context-free grammar to define the syntactic structure of a Jumbala program. The structure of the grammar is similar to that presented for Java [2], but our notation is different.

A grammar rule has a left hand side, which is a nonterminal symbol, and a right hand side, which contains zero or more terminal or nonterminal symbols. Nonterminal symbols are typeset in *Italics*. Terminal symbols are

typeset in `typewriter` face when they represent characters as they appear in a Jumbala program. The terminal symbols whose appearance varies in the source program are typeset in SMALL CAPITAL letters.

The following two example illustrate grammar rules.

Program ::=
    $\big($*TypeDeclaration* | *BlockStatement*$\big)^*$

TypeName ::=
    $\big[$*TypeName* `.` $\big]$ IDENTIFIER

The left and right hand sides are separated by the characters ::=. We use special characters in the right hand sides to make the notation more succinct. They are explained below in the order of decreasing precedence.

- The Kleene star * denotes zero or more occurrences of the element immediately preceding it. Do not confuse with the terminal symbol `*`, which denotes an asterisk in the input program.
- If elements are only separated by space, they are concatenated to form a new element. In terms of precedence, concatenation is stronger than the | operator but weaker than the Kleene star.
- A vertical line | separates two alternative elements, which is a shorthand notation for two separate grammar rules. Do not confuse with the terminal symbol `|`.
- An element inside square brackets $\big[\ \big]$ is optional, i.e. there may be zero or one occurrences of it. Do not confuse with literal square brackets `[` and `]`.
- Parentheses $\big(\big)$ are used to group together elements of the right hand side. Parentheses are only used to affect the precedence of special symbols. If parentheses were omitted in the first example above, the Kleene star would only affect the symbol *BlockStatement*. Do not confuse with the terminals symbols `(` and `)`.

The nonterminal symbol *Program* is special in that it is the start symbol of the grammar. Appendix B contains a list of all the grammar rules.

Chapter 2

Lexical Structure

2.1 Input Elements

A Jumbala *program* is a string of 8-bit characters. The significant part of the program may only contain 7-bit ASCII characters. All characters outside the 7-bit range are considered special characters, which may appear only inside comments and string literals. Unlike Java, Jumbala does not support Unicode characters.

The string of input characters is grouped into a sequence of input elements. An input element can be a *white space* character, a *comment*, or a *token*. White space and comments do not contribute to the semantics of a program, other than by separating tokens from each other.

Tokens are the terminal symbols of the syntactic grammar. A token can be either an identifier, a keyword, a literal, a separator, or an operator.

Program lines must end with the *newline* character (ASCII 10). Other line terminators are not recognized. In particular, carriage returns (ASCII 13) are not allowed (as they are in Java), except in comments and string literals. A newline is not required after the last line of the program.

Newlines are treated as white space. Other white space characters are the horizontal tab (ASCII 9), form feed (ASCII 12), and space (ASCII 32).

2.2 Comments

Comments are parts of the input text that are ignored by the interpreter. There are two kinds of comments.

A *traditional comment* begins with the characters `/*` and ends with the next occurrence of the characters `*/`. A string of any characters, including

newlines, may appear in between. Therefore `/**/` is a comment but `/*/` is not. An unterminated traditional comment results in a compile-time error.

An *end-of-line comment* begins with the characters `//` and ends at the next newline character or at the end of the program, whichever comes first.

Comments do not nest, so `//` has no special meaning in a traditional comment, and conversely, `/*` and `*/` have no special meaning in an end-of-line comment.

Comments do not occur within string literals or any other tokens.

2.3 Identifiers

An *identifier* is a maximal unlimited-length sequence of the following characters: the underscore `_`, the dollar sign `$`, digits `0` to `9`, uppercase letters `A` to `Z`, and lowercase letters `a` to `z`. The first character of an identifier must not be a digit. An identifier cannot have the same spelling as a reserved word.

Two identifiers are the same if they are identical character sequences. Uppercase and lowercase letters are considered distinct.

The token associated with an identifier is IDENTIFIER. The token carries the name of the identifier as its value.

2.4 Reserved Words

The following character sequences are the reserved words of the language.

```
abstract        extends         null
assert          false           return
boolean         final           send
break           for             static
case            if              super
class           implements      switch
continue        instanceof      this
default         int             to
do              interface       true
else            native          void
enum            new             while
```

The words `false`, `null`, and `true` are *named literals*. Other reserved words are *keywords*.

The corresponding token has the same name as the reserved word, e.g. `else` or `true`.

2.5 Literals

A *literal* represents a null reference, a string, or a value of a primitive type in the source code.

Integer literals can be expressed in decimal (base 10), octal (base 8), or hexadecimal (base 16). The other literal types are boolean literals, string literals, and the null literal.

The reserved words `false` and `true` are the boolean literals. They represent the two values of the `boolean` type.

The reserved word `null` is the null literal, which represents the only value of the null type.

2.6 Integer Literals

Integer literals represent 32-bit integer numbers.

A decimal integer literal is either the single digit `0`, or a digit `1` to `9` followed by zero or more digits `0` to `9`. The value of the literal is the same as its interpretation as a 32-bit signed decimal number.

The largest decimal integer literal allowed is `2147483648` (2^{31}). The corresponding token is INT_LITERAL_MUST_NEGATE. It may only appear as the operand of the unary minus operator `-`. Decimal integers larger than this produce a compile-time error.

Decimal integers between `0` and `2147483647` may appear wherever an integer is allowed. They are represented by the token INT_LITERAL with an integer value.

An octal literal begins with the digit `0` that is followed by one or more digits `0` to `7`. This is interpreted as a signed 32-bit number. Octal literals `00` to `017777777777` represent non-negative numbers 0 to 2147483647, and `020000000000` to `037777777777` represent negative numbers -2147483648 to -1, respectively. Larger octal numbers are not allowed.

A hexadecimal literal has the prefix `0X` or equivalently `0x`, followed by one of more hexadecimal digits. Digits 0 to 9 are represented as such, and digits 10 to 15 are expressed as letters `A` to `F` or `a` to `f`. The interpretation is analogous to octal literals. Literals `0x0` to `0x7fffffff` represent non-negative numbers 0 to $2^{31} - 1$, and `0x80000000` to `0xffffffff` are the negative numbers -2^{31} to -1. Other hexadecimal numbers are not allowed.

An octal or hexadecimal literal always produces the token INT_LITERAL enclosed with the associated signed 32-bit integer value.

Java allows integer literals with the suffix `L`, but Jumbala does not. In other words, there are no long integer literals.

2.7 String Literals

A string literal is a sequence of zero or more *string characters* enclosed in double quotes (`"`). A string character can be either an *escape sequence* or any 8-bit input character except the double quote, backslash \, or newline.

An escape sequence is used to represent one character of the string by two or more input characters. It can be one of the following:

- `\b` (backspace, ASCII 8)
- `\t` (horizontal tab, ASCII 9)
- `\n` (newline, ASCII 10)
- `\f` (form feed, ASCII 12)
- `\r` (carriage return, ASCII 13)
- `\"` (double quote, ASCII 34)
- `\'` (single quote, ASCII 39)
- `\\` (backslash, ASCII 92)
- An *octal escape*, i.e. a backslash \ followed by an octal number between `0` and `377` (= 255 decimal). Leading zeros are allowed but the total amount of octal digits must be three or less. This escape sequence represents the character whose ASCII code is the value of the octal number.

Other escape sequences are not allowed.

A string literal produces the token STRING_LITERAL, whose value is the contents of the string without the double quotes.

2.8 Separators and Operators

The following 9 input characters are recognized as *separators*:

```
( ) { } [ ] ; , .
```

The following 37 character sequences are the *operators*:

```
=   >   <   !   ~   ?   :
==  <=  >=  !=  &&  ||  ++  --
+   -   *   /   &   |   ^   %   <<   >>   >>>
+=  -=  *=  /=  &=  |=  ^=  %=  <<=  >>=  >>>=
```

The name of the associated token is the same character sequence that forms the separator or operator.

Chapter 3

Programs

The string containing Jumbala source code is called the *program*.

Program ::=
    $\big($*TypeDeclaration* | *BlockStatement*$\big)^*$

BlockStatement ::=
    LocalVariableDeclarationStatement
  | *Statement*

A program consists of type declarations and block statements (statements and local variable declarations). Unlike in Java, statements may appear at top level, outside all type declarations as well as in methods and constructors. The block statements appearing at top level are *top-level statements*.

A program is executed by evaluating the top-level statements in sequence. There is no special `main` method like in Java. However, such a method may be defined and called explicitly from a top-level statement.

A top-level statement may refer to types, methods, and fields whose declaration appears before or after the top-level statement.

A local variable declaration that appears directly at top level declares a *top-level local variable*. Such a variable may only be referred to from top-level statements, not e.g. from methods, so it is not a global variable.

Programs may not import entities from other programs. There are no packages like in Java. A similar effect can be obtained using incrementality, explained below.

3.1 Incremental Programs

A program is *incremental* if it augments another program. The incremental program may refer to top-level types and top-level local variables of the

original program as if the two programs were concatenated as strings.

It is a compile-time error if an incremental program declares a top-level type with the same name as a top-level type in the original program, or a top-level local variable with the same name as a top-level local variable in the original program.

An incremental program may be executed only after the original program has finished executing. The state of top-level local variables and all class variables is preserved from the first execution.

Chapter 4

Types and Variables

Every variable and expression has a type that is known at compile time.

Type ::=
    PrimitiveType | *ReferenceType*

Types are divided to three categories. The *primitive types* are `int` and `boolean`. The *reference types* are all class, interface, enum, and array types. The special *null type* has only one value, `null`. The null type has no name inside Jumbala programs, so it cannot be the type of a variable.

4.1 Primitive Types and Values

The two *primitive types* are `int` and `boolean`, which work almost identically as their counterparts in Java.

PrimitiveType ::=
    `int` | `boolean`

The values of type `int` are 32-bit signed integers. The values of type `boolean` are `false` and `true`.

Values of primitive types are not objects and there are no references to primitive types. A variable or expression of a primitive type can only hold values of that exact type.

There are no character types (use `String`s instead), floating-point types, or other than 32-bit integer types.

Jumbala defines no boxed types like `Integer` or `Long` found in Java. The user can define classes with similar functionality.

4.2 Reference Types and Values

A reference type can be a class, interface, enum, or array type.

ReferenceType ::=
    ArrayType | *TypeName*

ArrayType ::=
    Type `[]`

Class, interface, and enum types are referred to by their names in programs (Section 5). An array type (Section 7) is denoted by its component type followed by an empty pair of square brackets.

A variable or expression of a reference type has a value that is either the `null` reference or a reference to an object of that type or one of its subtypes.

4.3 Objects and Memory

An *object* is an instance of either a class, enum, or array type. The type of an object is specified when creating the object and cannot be changed afterwards. The type cannot be an interface or an `abstract` class.

An object has identity and state. The state of an object consists of values for all instance variables that are members of the type of the object or one of its supertypes.

An object cannot be the value of a variable. However, several variables may hold a reference to an object at the same time.

Class and array instances are created using `new` expressions (Section 9.15 and 9.16) or array initializers (Section 7.1). Instances of an enum type cannot be created dynamically.

Jumbala pays no attention to the possibility of out-of-memory conditions. Object creation always succeeds from the language point of view. If memory runs out when creating objects or at any other point, it is considered an exceptional situation that halts the program but that cannot be observed from the program. In contrast, the Java language allows an out-of-memory exception (called OutOfMemoryError) to occur only at certain points in the program, and the error may be caught and handled within the program.

Objects have no scope that could be fixed at compile time. An object exists as long as it can be reached from the program by following references. An object is considered nonexistent when it becomes unreachable. It is not defined what happens then. A proper implementation has a garbage collector that removes unreachable data at some point. The effect of garbage collection

cannot be observed from the program because objects do not have finalizers like in Java.

4.4 Predefined Types

Jumbala does not have an extensive class library like Java. However, the definition of the language itself requires that a simple class structure exists. Therefore we define following five top-level types that exist implicitly in every Jumbala program.

4.4.1 The Class `Object`

`Object` is the primordial superclass of all other classes. It is a simplified version of `Object` in Java.

The class `Object` has no fields. Its only constructor is the default constructor, which creates a new object. It has the following methods, which are all non-`final` instance methods.

- `clone`, which takes no parameters and returns an `Object`. The method results in a run-time error if it is invoked for an object whose class is not a subclass of `Cloneable`. For example, `new Object().clone()` evaluates to a run-time error. If the method is invoked on a `Cloneable` object, it returns a reference to a new object whose fields have the same values as the fields of the original object, i.e. a shallow copy of the original object.

- `equals`, which takes one parameter of type `Object` and returns a `boolean`. The method returns `true` if its argument refers to the same object that the method is invoked on, otherwise `false`. The method may be overridden to make semantic comparison between two objects.

- `toString`, which takes no parameters and returns a reference to a `String`. The returned string is defined by the implementation, and it may not be `null`. This method may be overridden to produce useful human-readable representations of objects.

4.4.2 The Interface `ObjectInterface`

`ObjectInterface` is a subtype of `Object` and a supertype of all interfaces. It exists explicitly in Jumbala, although it does not have a named counterpart

in Java. The effect is the same in both languages: all methods of class `Object` are accessible through a reference of any interface type.

`ObjectInterface` has no fields and its methods have the same names, parameter types and return types as `Object`. The methods in `ObjectInterface` differ from the methods in `Object` in tha they are `abstract`.

4.4.3 The Interface `Cloneable`

The `Cloneable` interface is a subtype of `ObjectInterface`. It does not have any fields and its methods are those inherited from `ObjectInterface`.

To make a class whose objects can be cloned, declare the class as a subtype of `Cloneable` and, if necessary, override the `clone` method to first invoke `super.clone` and then perform other necessary copying activities, just like in Java.

4.4.4 The Class `String`

`String` is a `final` class representing a string of text. It is the class of string literals (Section 2.7). The members of `String` are defined in the implementation.

4.4.5 The Class `Enum`

`Enum` is an `abstract` class that acts as the common supertype of all enum types. Although `Enum` is not `final`, it is a compile-time error to declare a subclass of `Enum`. The methods of `Enum` are those inherited from `Object`, and it has not fields.

4.5 Subtyping

The *subtype* relationship is a reflexive, transitive, antisymmetric relationship between types. Every type is a subtype of itself. If T is a subtype of S, then S is a *supertype* of T. The relationship is characterized by the following rules.

A primitive type has no subtypes or supertypes besides itself.

The null type is a subtype of all reference types. No primitive or reference type is a subtype of the null type.

The subtype relationship between two reference types is determined by the *direct supertype* relationship. Type T is a subtype of S if and only if

there exists a chain of types $T = T_1,\ T_2,\ \ldots,\ T_n = S$ such that each T_{k+1} is a direct supertype of T_k.

The class `Object` has no direct supertypes.

The direct supertypes of a class other than `Object` are its direct superclass and direct superinterfaces. The direct superclass is the one mentioned in the class declaration (after the keyword `extends`), or `Object` if no class is explicitly mentioned. The direct superinterfaces are those mentioned in the class declaration (after the keyword `implements`), if any.

The direct supertype of `ObjectInterface` is `Object`.

The direct supertypes of an interface other than `ObjectInterface` are its direct superinterfaces, i.e. those mentioned in the interface declaration. If none are mentioned, the only direct supertype if `ObjectInterface`.

The direct supertype of an enum type is the `abstract` class `Enum`.

The direct supertypes of an array of either a primitive type or the class `Object` are `Object` and the interface `Cloneable`.

The direct supertypes of T`[]`, an array of a reference type T other than `Object`, are the array types S`[]` where S is a direct supertype of T.

It follows from these rules that `Object` is a supertype of all reference types. `ObjectInterface` is a supertype of all interfaces. `Object[]` and `Cloneable` are supertypes of all array types.

4.6 Castability

We say that type T is *castable* to type U if, informally, it is possible that a variable of type T has the same value as a variable of type U. Castability is a reflective, symmetric, and transitive relation between types.

T is castable to U if any of the following apply:

- One of T, U is a subtype of the other.
- One of T, U is a non-`final` class type and the other is an interface.
- Both T and U are array types and the component type of T is castable to the component type of U.

The Java specification [2] talks about casting conversions and not castability. Jumbala has no conversions between values, so the simpler notion of castability is introduced.

4.7 Variables

A variable is a typed storage location. A variable of a primitive type can hold a value of that type. A variable of a reference type can hold either a reference to an object of that type or one of its subtypes, or the `null` reference.

4.7.1 Kinds of Variables

There are six kinds of variables in Jumbala. These correspond to the kinds of variables in Java, with the omission of exception-handler parameters.

1. A *class variable* is a field declared using the keyword `static` (Section 6.6).
2. An *instance variable* is a field declared without the keyword `static` (Section 6.6).
3. An *array component* is one of the storage locations in an instance of an array (Section 7). It has no name.
4. A *method parameter* variable is created for each declared parameter when a method is invoked. The parameter initially has the argument value given in the method invocation (Section 9.14).
5. A *constructor parameter* is similar to a method parameter, but is associated with the creation of a class instance (Section 9.15).
6. A *local variable* is declared by a local variable declaration (Section 8.2).

Class and instance variables may be declared `final`, so that they cannot be assigned to after initialization. Unlike Java, Jumbala does not allow method or constructor parameters or local variables to be `final`.

4.7.2 Default Values

Every class variable, instance variable, and array component is initialized to a default value when it is created. Local variables are not automatically initialized to default values.

For a variable whose type is `int`, the default value is `0`.

For a variable whose type is `boolean`, the default value is `false`.

For a variable whose type is a reference type, the default value is `null`.

Chapter 5

Names

A *name* is a textual reference to a declared entity in a program. Names are classified to

- *type names*, which resolve to class, interface, and enum types,
- *method names*, which are used in method invocations, and
- *variable names*, which resolve to fields, local variables, and method or constructor parameters.

There are no package names in Jumbala. What we call variable names are called expression names in the Java specification.

A name can be *simple*, consisting of a single identifier, or *qualified*, consisting of several identifiers separated with periods.

TypeName ::=
    [*TypeName* .] IDENTIFIER

MethodName ::=
    [*AmbiguousName* .] IDENTIFIER

VariableName ::=
    [*AmbiguousName* .] IDENTIFIER

AmbiguousName ::=
    [*AmbiguousName* .] IDENTIFIER

The classification of names is mostly handled by the grammar rules. The only place in which a name is ambiguous according to the grammar is when

a variable name or a method name is qualified, for example in the expressions `Date.year` or `Date.reset()`. It is unclear, without seeing the entire program, whether `Date` refers to a type or a variable, so the left part of the name is first classified as an *AmbiguousName*. An ambiguous name is reclassified by the rules explained in Section 5.5.

An identifier used in a declaration to define the name of an entity is not considered to be a name itself. In the code fragment below, the identifier `flag` on the first line is not a name, but on the second line it is.

```
boolean flag;
flag = true;
```

5.1 Scope

The accessibility of an entity is affected by the *scope* of the declaration of the entity. There are no explicit access modifiers such as `private` or `public` in Jumbala; everything is implicitly public.

Scope is a static, compile-time concept. Declarations have scopes, objects do not.

Scoping does not entirely determine what is accessible or not. It may be possible to directly access a variable that is out of scope, for example, using a qualified name. Conversely, it might not be possible to access a variable or type that is in scope because it is shadowed or obscured by another declaration. The concepts of shadowing and obscuring are explained in the Java specification, and they are also present in Jumbala. They are not described here explicitly, as they are logical consequences of the rules of name resolution.

The scopes of different kinds of declarations are given below.

The scope of a top-level local variable declaration is the entire program and the following incremental programs.

The scope of a local variable declaration that is not a top-level declaration and not in the initialization part of a `for` statement is the rest of the block, method/constructor body, or switch block that directly encloses the declaration.

The scope of a local variable declaration in the initialization part of a `for` statement is the rest of the for statement, including the body.

The scope of a method or constructor parameter is the body of the method or constructor.

5.2 Resolution of Type Names

A type name resolves to a class, interface, or enum type.

5.2.1 Simple Type Names

A simple type name N is resolved as follows. If the name appears in the body of a type declaration, let $T_1, \ldots, T_n$ be the enclosing types such that T_{i+1} is nested in T_i.

If there exists an index i such that T_i has at least one (declared or inherited) member type named N or the simple name of T_i itself is N, then let k be the largest such index. If T_k has exactly one member type named N, the name resolves to that type. If T_k has more than one member type named N, a compile-time error occurs. If T_k has no member type named N, the name resolves to T_k.

If no such index exists and there is a top-level type named N, the name resolves to that type. If there is no top-level type named N, a compile-time error occurs.

5.2.2 Qualified Type Names

A qualified type name $Q.N$, where Q is a type name and N is an identifier, is resolved as follows. First Q is resolved. If this does not result in a compile-time error, Q resolves to a type T. If T has zero or more than one member types named N, a compile-time error occurs. Otherwise $Q.N$ resolves to the unique member type named N.

5.3 Resolution of Method Names

A method name can only appear in a method invocation expression. The first part in resolving a method name is to locate a reference type to search for methods. That part is explained below. The entire method invocation procedure is explained in Section 9.14.

5.3.1 Simple Method Names

A simple method name N must appear in the body of a type declaration, or a compile-time error occurs. Let $T_1, \ldots, T_n$ be the enclosing types such that T_{i+1} is nested in T_i.

Let k be the largest index i such that T_i has at least one (declared or inherited) method named N. If no such index exists, a compile-time error occurs. The type to search is T_k.

5.3.2 Qualified Method Names

A qualified method name has the form $Q.N$ where Q is an ambiguous name and N is an identifier. Q is first reclassified.

If Q is reclassified as a variable name, the type to search is the type of the variable that Q resolves to.

If Q is reclassified as a type name, the type to search is the type that Q resolves to.

5.4 Resolution of Variable Names

A variable name resolves to a variable. Variable names only appear in the context of expressions. A variable name can be viewed as a special kind of expression. It has a type and it can be evaluated at run-time.

The result type of a variable name is the type of the variable that the name resolves to. The result value is the value of the variable. A variable name as an expression is an lvalue if and only if it resolves to a non-`final` variable.

5.4.1 Simple Variable Names

A simple variable name N is resolved as follows. If the name appears in the scope of a local variable, method parameter, or constructor parameter named N, the name resolves to that variable. If the variable is a local variable that is uninitialized (Section 8.2) when the variable name is evaluated, a run-time error occurs.

Assume that there is no local variable or method/constructor parameter named N in scope. If N is not enclosed in the body of a type declaration, a compile-time error occurs. Otherwise let $T_1, \ldots, T_n$ be the enclosing types such that T_{i+1} is nested in T_i.

Let k be the largest index i such that T_i has at least one (declared or inherited) field named N. If no such index exists, a compile-time error occurs. If T_k has more than one field named N, a compile-time error occurs. Otherwise T_k has exactly one field named N.

If the field is `static`, the name resolves to that field. Assume that the field is an instance variable. If $k \neq n$ or the name appears in a static context

(Section 9.1), a compile-time error occurs. Otherwise the name resolves to the instance variable N (declared in T_k) in the current object (Section 9.1).

5.4.2 Qualified Variable Names

A qualified variable name has the form $Q.N$ where Q is an ambiguous name and N is an identifier. Q is first reclassified.

If Q is reclassified as a variable name, it resolves to a variable V with type T. T must be a reference type with exactly one field named N, or a compile-time error occurs. If the field is `static`, $Q.N$ resolves to that field. If the field is an instance variable, $Q.N$ resolves to that variable in the object referred to by V. In the latter case, if V evaluates to `null` at run-time, a run-time error occurs.

If Q is reclassified as a type name, it resolves to a type T. T must have exactly one field named N that is a `static`, or a compile-time error occurs. $Q.N$ resolves to that field.

5.5 Reclassification of Ambiguous Names

An ambiguous name is reclassified to a type name or variable name that is resolved by the rules stated above. The reclassification always succeeds but the consequent resolution may result in a compile-time error.

5.5.1 Simple Ambiguous Names

A simple ambiguous name N is reclassified as follows. If the name appears in the scope of a local variable, method parameter, or constructor parameter named N, the name is reclassified as a simple variable name.

Assume that there is no local variable or method/constructor parameter named N in scope. If N is not enclosed in the body of a type declaration, the name is reclassified as a simple type name. Otherwise let $T_1, \ldots, T_n$ be the enclosing types such that T_{i+1} is nested in T_i.

If an index i exists such that T_i has at least one (declared or inherited) field named N, the name is reclassified as a simple variable name. If not, the name is reclassified as a simple type name.

5.5.2 Qualified Ambiguous Names

A qualified ambiguous name has the form $Q.N$ where Q is an ambiguous name and N is an identifier. Q is first reclassified.

If Q is reclassified as a variable name, $Q.N$ is reclassified as a variable name.

Assume Q is reclassified as a type name. If the resolution of type name Q does not produce a compile-time error and if Q resolves to a type that has at least one field named N, then $Q.N$ is reclassified as a variable name. Otherwise $Q.N$ is reclassified as a type name.

Chapter 6

Type Declarations

Type declarations define new reference types.

TypeDeclaration ::=
    ClassDeclaration
  | *InterfaceDeclaration*
  | *EnumDeclaration*

6.1 Top-Level Types

A type declaration that appears at top level in a program declares a top-level type.

It is a compile-time error if two top-level types with the same name are declared in the same program or in two different programs, one of which increments the other. It is a compile-time error if a top-level type is declared with the same name as a predefined type (Section 4.4).

6.2 Nesting in Types

The declaration of a class or interface T has a body that can contain further declarations. Any declaration, name, or expression E that appears within the body of T is said to be *enclosed* in declaration of T. T is the *directly enclosing* type of E if there is no type enclosed in T that also encloses E.

A type whose declaration is enclosed in T is a *nested type* of T. All nested types in Jumbala are implicitly `static`. This is a deviation from Java, where a nested class that is not declared using the keyword `static` can access the instance variables of enclosing classes.

It is a compile-time error if two type declarations with the same directly enclosing type declare a type with the same name.

6.3 Class Declarations

A class declaration defines a class, its name, supertypes, members, and constructors.

ClassDeclaration ::=
    [`abstract` | `final`] `class` IDENTIFIER
        [`extends` *TypeName*]
        [`implements` *TypeName* (`,` *TypeName*)*]
        `{` *ClassBodyDeclaration** `}`

ClassBodyDeclaration ::=
    MemberDeclaration
  | *ConstructorDeclaration*

MemberDeclaration ::=
    FieldDeclaration
  | *MethodDeclaration*
  | *TypeDeclaration*

The IDENTIFIER following the keyword `class` is the name of the class.

6.3.1 Class Modifiers

The modifier `abstract` or `final` (but not both) may appear in the beginning of the declaration.

The class is an `abstract` class if it is declared using the keyword `abstract`. An `abstract` class cannot be instantiated but it may have `abstract` methods.

The class is a `final` class if it is declared using the keyword `final`. A `final` class may not have subclasses.

6.3.2 Direct Supertypes

If the keyword `extends` followed a type name appears in the declaration, that type name is resolved (Section 5.2) to the *direct superclass* of the class. It is a compile-time error if the direct superclass is not a class, or if it is a `final` class, or if it is the class `Enum`. If the `extends` part is missing, the direct superclass of the class is `Object`. The class `Object` itself has no direct superclass.

The direct superinterfaces of the class, if any, are specified after the keyword `implements`. The type names following `implements` must all resolve to interface types, no two of which are allowed to be the same type. Otherwise a compile-time error occurs.

It is a compile-time error if one of the type names after the words `extends` or `implements` resolves to a type whose declaration either encloses the class declaration or appears textually after the class declaration. This is a notable difference from Java, which allows a subtype to be declared before its supertypes.

The direct superclass and direct superinterfaces are the direct supertypes of the class, as explained in (Section 4.5).

6.4 Interface Declarations

An interface declaration defines an interface type.

InterfaceDeclaration ::=
    `interface` IDENTIFIER
        [`extends` *TypeName* (`,` *TypeName*)*]
        `{` *MemberDeclaration** `}`

MemberDeclaration ::=
    FieldDeclaration
  | *MethodDeclaration*
  | *TypeDeclaration*

The IDENTIFIER following the keyword `class` is the name of the class. Unlike classes, interfaces cannot be declared using modifiers. It can be thought that an interface is always implicitly `abstract` and `static` and never `final`.

6.4.1 Direct Supertypes

If the keyword `extends` followed one or more a type names appears in the declaration, that type names are resolved (Section 5.2) to the *direct superinterfaces* of the interface. It is a compile-time error if two of the names resolve to the same type of if not all of the types are interfaces. If the `extends` part is missing, the only direct superinterface is `ObjectInterface`. The interface `ObjectInterface` has one direct supertype, `Object`.

It is a compile-time error if the declaration of any of the superinterfaces either encloses the interface declaration or appears textually after the interface declaration. This is a notable difference from Java, which allows a subtype to be declared before its supertypes.

The direct superinterfaces are the direct supertypes of the interface, as explained in (Section 4.5).

6.5 Enum Declarations

An enum declaration defines an enumerated type.

EnumDeclaration ::=
`enum` IDENTIFIER `{` [IDENTIFIER (`,` IDENTIFIER)*] [`,`] `}`

Enumerated types or enums are types that have a fixed, finite set of values known as *enum constants*. New instances of an enum type may not be created dynamically. Enums in Jumbala are less powerful than in Java. Enum constants are not allowed to have internal structure. Their only attributes are their names and identities. Enum constants are objects and references of enum types can be compared using operators `==` and `!=`.

The IDENTIFIER appearing after the keyword `enum` is the name of the enum type. The identifiers inside curly braces are the names of the enum constants. Enum constants of a single type must have distinct names.

The only direct supertype of an enum is the class `Enum`.

6.5.1 Members of an Enum

The members of an enum type are the fields defined by enum constants in the enum declaration, the methods inherited from class `Object`, and the method `values`.

Every enum constant appears as a `final static` field in the enum type. The type of the field is a reference to the enum type, and the value of the field is a reference to the object representing the enum constant. The fields

of an enum are initialized before the initialization of `static` fields in classes and interfaces, so they can never be observed to have the value `null`. An enum constant named `c` in an enum type `E` is accessed using the expression `E.c`.

The instance method `toString` of an enum type returns a string representation of an enum constant. The representation is implementation dependent. The instance method `clone` results in a run-time error if executed. The instance method `equals` takes a parameter of type `Object` and returns `true` if and only if the argument is a reference to the same object the method is invoked on. The class method `values` of an enum type `E` takes no parameters and returns a reference to an array of `E`. The components of the returned array are references to the enum constants of `E` in the same order in which they are listed in the enum declaration.

There are no methods in an enum to directly support ordinal numbers for enum constants (c.f. the method `ordinal` in the Java class `Enum`).

An enum type has no member types.

6.6 Field Declarations

A field declaration declares one or more fields for the directly enclosing class or interface.

FieldDeclaration ::=
    (`static` | `final`)* *Type*
        VariableDeclarator (`,` *VariableDeclarator*)* `;`

VariableDeclarator ::=
    IDENTIFIER [`=` *VariableInitializer*]

VariableInitializer ::=
    Expression | *ArrayInitializer*

ArrayInitializer ::=
    `{` [*VariableInitializer* (`,` *VariableInitializer*)*] [`,`] `}`

All fields declared by a single declaration have the same type and modifiers. The modifiers `static`, `final`, or both can be used. If the same modifer appears twice, a compile-time error occurs.

A field declared with the modifier `static` is a class variable. There is exactly one instance of a class variable at run-time, and it is not associated

with an object. If the word `static` is not used, the field is an instance variable.

A field declared with the modifier `final` cannot be assigned to. An attempt results in a compile-time error. A `final` field must have an initializer. In contrast, Java allows a `final` field not to have an initializer if the field is assigned exactly once in each constructor.

It is a compile-time error if two fields with the same name are declared in the same class or interface. However, a class or interface may have two fields with the same name as members by inheritance.

6.6.1 Initialization of Fields

Unlike local variables, fields cannot be left uninitialized because they are initialized to default values.

All class variables are initialized to their default values (Section 4.7.2) depending on their type before any statements of the program are executed. After that, for each variable initializer of a static variable, the initializer is evaluated and the result is assigned to the variable. Initializers are evaluated in the order in which they appear in the program. All this happens before any top-level statements of the program are executed. However, initialization of static variables in an incremental program takes place only after the previous programs have been executed.

In the following example, fields `w` and `x` get the value 0, and `y` and `z` get the value 10.

```
class C {
  static int w;
  static int x = y;
  static int y = 10;
  static int z = y;
}
```

The initialization of instance variables is specified in Section 9.15.

In Java there are restrictions on whether field initializers may refer to other fields. Also, the order of initialization is different if fields are initialized with compile-time constants. These special cases do not apply to Jumbala.

6.7 Method Declarations

A method declaration declares a method of the directly enclosing class or interface.

MethodDeclaration ::=
    (`abstract` | `static` | `final` | `native`)* (*Type* | `void`)
        IDENTIFIER *FormalParameters* *MethodBody*

FormalParameters ::=
    `(` [[`final`] *Type* IDENTIFIER (`,` [`final`] *Type* IDENTIFIER)*] `)`

MethodBody ::=
    `{` *BlockStatement** `}`
  | `;`

Modifiers `abstract`, `static`, `final`, or `native` may be used in the declaration. It is a compile-time error if the same modifier appears twice. All methods of interfaces are considered `abstract`, whether the modifier is used or not.

An `abstract` method has no body and cannot be directly invoked. An `abstract` method cannot be `static`, codefinal, or `native`, or a compile-time error occurs.

If the word `static` is used, the method is a class method. Otherwise it is an instance method.

If a method is declared `final`, it cannot be hidden or overridden in a subclass.

A `native` method is one that is no implemented in Jumbala but some native programming language. A `native` method is invoked by the same rules as an ordinary method. The details are implementation-specific.

A method may have any number of formal parameters. Variable arity methods are not supported. When the method is invoked, a variable is created for each formal parameter, and the argument values are assigned to the variables. The parameter variables can be accessed from the method body. A `final` parameter cannot be assigned to. It is a compile-time error if a method has two formal parameters with the same name.

A method must have a body with zero or more block statements if and only if the method is not `abstract` and not `native`. If the method is `abstract` or `native`, the body must be replaced by a semicolon.

It is a compile-time error if two methods with the same name and same parameter types are declared in the same class or interface. Further restrictions on methods with respect to subtyping are given in Section 6.9.3.

6.8 Constructor Declarations

A constructor declaration declares a constructor for the directly enclosing class.

ConstructorDeclaration ::=
IDENTIFIER *FormalParameters* *ConstructorBody*

FormalParameters ::=
`(` [[`final`] *Type* IDENTIFIER (`,` [`final`] *Type* IDENTIFIER)*] `)`

ConstructorBody ::=
`{` [`super` *Arguments* `;`] *BlockStatement** `}`

Arguments ::=
`(` [*Expression* (`,` *Expression*)*] `)`

The IDENTIFIER at the beginning of a constructor declaration must be the simple name of the directly enclosing class, or a compile-time error occurs.

Constructors, like methods, have formal parameters. No two formal parameters of a constructor may have the same name. Argument values are assigned to constructor parameters upon constructor invocation (Section 9.15). It is a compile-time error if a class has two constructors with the same number and types of parameters.

Constructors are not members. They are never inherited, hidden, or overridden.

The body of a constructor may begin with an explicit superclass constructor invocation using the keyword `super` with associated argument expressions. If there is no explicit superclass constructor invocation, an implicit superclass constructor invocation is assumed. The implicit invocation is equivalent to an invocation with no parameters, i.e. `super();`. In this case it is a compile-time error if the direct superclass does not have a constructor that takes no parameters. The semantics for superclass constructor invocation is given in Section 9.15.

It is not possible to invoke an alternate constructor of the same class in the beginning of a constructor, like in Java.

6.9 Members of Classes and Interfaces

The members of a class or interface type are the fields, methods, and member types that are either declared in the class or interface declaration or inherited from direct supertypes. Constructors are not members.

A type may have several kinds of members with the same name as illustrated below.

```
class Confuse {
  Confuse() {}      // A constructor
  void Confuse() {} // A method
  void Thing() {}   // A method
  class Thing {}    // A nested class
  Thing Thing;      // A field
}
```

6.9.1 Member Types

If a class or interface type T declares a nested type U named N, the declaration *hides* any types named N that are members of the direct supertypes of T. U is a member of T but the hidden types are not members of T.

A class or interface type T inherits a type U if U is a member of a direct supertype of T and if T declares no type with the same name. The inherited type is a member type of T.

A type may declare at most one nested type with a given name. However, a type may have several member types with the same name as inherited types. Trying to access such a type by its simple name is an error (Section 5.2).

A type may inherit the same member type from two different direct supertypes if they have a common supertype that declares the nested type. The member type is considered to be inherited only once, and it may be accessed by its simple name.

6.9.2 Fields

A field is either an instance variable (non-`static`) or class variable (`static`).

If a class or interface type T declares a field F named N, the declaration *hides* any fields named N that are members of the direct supertypes of T. The hidden fields may have a different type than F. Both instance and class variables can hide instance variables and class variables. F is a member of T but the hidden variables are not members of T.

Even if an instance variable H is hidden, an object of type T (if T is not an interface) has a value for the variable H. The variable can be accessed by the expression `super.`N (Section 9.12) or by `((`*supertype*`) this).`N.

Hidden class variables can be accessed by the expression *supertype*`.`N.

A class or interface type T inherits a field F if F is a member of a direct supertype of T and if T declares no field with the same name. The inherited field is a member of T.

A type may declare at most one field with a given name. However, a type may have several fields with the same name as inherited members. Trying to access such a field by its simple name is an error (Section 5.4).

A type may inherit the same field from two different direct supertypes if they have a common supertype that declares the field. The field is considered to be inherited only once, and it may be accessed by its simple name.

6.9.3 Methods

In the context of inheritance, methods are compared to each other using their signatures, not only their names. The signature of a method consists of the name of the method, the number of parameters it takes, and the ordered list of the types of parameters.

Let M_1 and M_2 be two methods with the same signature. We say that M_1 *conforms* to M_2 if i) the return type of M_1 is a subtype of the return type of M_2, if ii) M_1 and M_2 are either both class methods or both instance methods, and if iii) M_2 is not `final`.

Overriding by Instance Methods. If a class or interface type T declares an instance method M (`abstract` or not) with a signature S, the declaration *overrides* any methods with signature S that are members of the direct supertypes of T. Method M is a member of T but the overridden methods are not. M must conform to each method with signature S that is a member of any of the direct supertypes of T, or a compile-time error occurs. An instance method that is not `abstract` can be overridden by an `abstract` method, and vice versa.

Hiding by Class Methods. If a class or interface type T declares a class method M with a signature S, the declaration *hides* any methods with signature S that are members of the direct supertypes of T. Method M is a member of T but the hidden methods are not. It is a compile-time error if M does not conform to all of the hidden methods.

Inheritance. Assume that a class or interface T has a direct supertype that has a method with signature S but T does not declare a method with signature S. Let $M_1, \ldots, M_n$ be the methods with signature S in the direct supertypes. At most one of the methods is non-`abstract` because only one of the direct supertypes can be a class. Type T inherits one of the methods by the following rules.

1. If one of the methods is a class method, then it is a compile-time error if $n > 1$. Type T inherits the class method.

2. If one of the methods, say M_i, is a non-`abstract` instance method, it is a compile-time error if it does not conform to every other M_j. T inherits M_i.

3. If all methods are `abstract`, it is a compile-time error if T is a non-`abstract` class. One of the methods, say M_i, must conform to all others, or a compile-time error occurs. Type T inherits M_i. There may be several methods that conform to all others but the choice is arbitrary and does not affect the semantics.

The Java specification defines case 3 above differently. An `abstract` class or interface can have several `abstract` methods with the same signature as members. Our approach produces the same semantics.

Chapter 7

Arrays

Arrays are the only built-in collection type in Jumbala. Arrays in Jumbala are mostly identical to those in Java.

For each type T there exists implicitly an array type T`[]` whose component type is T. Array types are reference types, and arrays themselves are objects. The *length* of an array (the number of components in it) is determined when the array is created. The components of an array are unnamed variables whose type is the component type of the array. Components are numbered from 0 to $n - 1$, where n is the length. Multidimensional arrays can be simulated by arrays of arrays.

The componentes or an array are accessed using array access expressions (Section 9.13). The components are never `final` variables. When an array is created using an array creation expression, the values of the components are the default values (Section 4.7.2) of the component type.

7.1 Array Initializers

An array is created either using an array creation expression (Section 9.16) or when initializing a variable with an array initializer.

ArrayInitializer ::=
`{` [*VariableInitializer* (`,` *VariableInitializer*)*] [`,`] `}`

VariableInitializer ::=
Expression | *ArrayInitializer*

The variable being initialized must be of an array type, call it T`[]`, and the types of expressions appearing in the initializer must be subtypes of T. When the array initializer is evaluated, the expressions are evaluated left to

right. The result of an array initializer is an array of T whose component values are the results of the expressions.

7.2 Members of an Array

Arrays are cloneable objects, and every array type is a subtype of `Object` (see Section 4.5 for details).

An array has one field as its member. The field is a `final` instance variable of type `int` and its name is `length`. The value of the field is the number of components in the array.

The methods of an array type are those inherited from `Object`, except `clone`. The `clone` method of an array is overridden to return a new array whose component values are copied from the original array. The return type of the method is the array type.

Arrays have no member types.

Chapter 8

Statements

Statements are elements that constitute executable code. A statement can be executed to produce an effect. Statements may contain other statements and expressions. Statements are also used to control program flow.

The set of statements in Jumbala is principally the same as in Java, with some simplifications.

We use the term block statement to denote statements and local variable declarations. Block statements may appear in method bodies, constructor bodies, the top level of a program, and in blocks.

BlockStatement ::=
    LocalVariableDeclarationStatement
  | *Statement*

Statement ::=
    StatementWithoutTrailingSubstatement
  | *LabeledStatement*
  | *IfThenStatement*
  | *IfThenElseStatement*
  | *WhileStatement*
  | *ForStatement*

StatementNoShortIf ::=
    StatementWithoutTrailingSubstatement
  | *LabeledStatementNoShortIf*
  | *IfThenElseStatementNoShortIf*
  | *WhileStatementNoShortIf*
  | *ForStatementNoShortIf*

StatementWithoutTrailingSubstatement ::=
    Block
  | *EmptyStatement*
  | *ExpressionStatement*
  | *SendStatement*
  | *AssertStatement*
  | *SwitchStatement*
  | *DoStatement*
  | *BreakStatement*
  | *ContinueStatement*
  | *ReturnStatement*

For rationale behind *StatementNoShortIf*, see Section 8.7.

8.1 Blocks

A block is a composition of zero or more statements, enclosed in braces.

Block ::=
    `{` *BlockStatement** `}`

BlockStatement ::=
    LocalVariableDeclarationStatement
  | *Statement*

Blocks are used to group together statements, for example, in the branches of an `if` statement. They also define a scope for local variables (Section 5.1).

When a block is executed, the contained statements are executed in sequence.

8.2 Local Variable Declaration Statements

A local variable declaration statement declares one or more local variables.

LocalVariableDeclarationStatement ::=
    LocalVariableDeclaration `;`

LocalVariableDeclaration ::=
    [`final`] *Type* *VariableDeclarator* (`,` *VariableDeclarator*)*

VariableDeclarator ::=
IDENTIFIER [= *VariableInitializer*]

Local variables can be declared in blocks, in method or constructor bodies, in the initialization of a `for` statement, or at top level. A local variable has a limited scope (Section 5.1).

All local variables declared in a single statement have the same type. A local variable can be `final`, in which case it cannot be assigned to after initialization, or a compile-time error occurs. In Jumbala, all `final` variables must have an initializer.

The name of a local variable is defined by the IDENTIFIER in its declarator. It is a compile-time error if a local variable in a method or constructor body has the same name as a formal parameter (Sections 6.7 and 6.8). It is a compile-time error if a local variable is declared in the scope of another local variable with the same name.

Unlike any other kinds of variables, local variables are *uninitialized* by default. Accessing an uninitialized local variable results in a run-time error, unless the access is a simple assignment to the variable. When a local variable declaration is executed, its variable declarators are executed left to right. When a local variable declarator is executed, the local variable is created and its value is first uninitialized. If the declarator has an initializer, it is evaluated, and the result is assigned to the variable. The variable is in scope in its own initializer and in any subsequent initializers.

There is one way to enter the scope of a local variable without executing the declaration, as illustrated below. In this case, any variable initializers are not evaluated.

```
switch (10) {
case 9:
  int x = 12;   // Not reached
  break;
case 10:
  // Variable x is in scope, but uninitialized.
  System.out.println(x);   // Run-time error.
  x = 10;   // Ok.
  break;
}
```

8.3 The Empty Statement

The empty statement consists of a single semicolon. It carries no action when executed.

EmptyStatement ::=
    ;

8.4 Labeled Statements

Labeled statements are used to direct the control flow when using labeled `break` and `continue` statements.

LabeledStatement ::=
    IDENTIFIER : *Statement*

LabeledStatementNoShortIf ::=
    IDENTIFIER : *StatementNoShortIf*

It is a compile-time error if a labeled statement is enclosed in another labeled statement with the same identifier as the label.

8.5 Expression Statements

An expression statement consists of an expression followed by a semicolon. The statement evaluates the expression and discards the result.

Only certain kinds of expressions are allowed as the top-level expression in an expression statement. These are simple or compound assignment, pre- or postincrement expressions, and pre- or postdecrement expressions, method invocation, and class instance creation expression.

ExpressionStatement ::=
    StatementExpression ;

StatementExpression ::=
    Assignment
  | *PreIncrement*
  | *PreDecrement*
  | *PostIncrement*
  | *PostDecrement*
  | *MethodInvocation*
  | *ClassInstanceCreation*

8.6 The `send` Statement

The `send` statement is a construct that does not exist in Java at all. Its purpose is to model asynchronous transmission of a signal.

SendStatement ::=
`send` IDENTIFIER *Arguments* `to` *Expression* `;`

Arguments ::=
`(` [*Expression* (`,` *Expression*)*] `)`

The semantics of a `send` statement is defined by the semantics of a method invocation statement (Section 9.14). A `send` statement is equivalent to the following expression statement.

`(` *Expression* `).$$signal_`IDENTIFIER `(` *Arguments* `);`

The latter statement is an invocation of a method whose name is the concatenation of the string '`$$signal_`' and the name of the signal, i.e. the IDENTIFIER in the `send` statement. (Dollar signs are valid characters in identifiers.) Usual compile-time and run-time checks apply. For example, the type of the *Expression*, which denotes the object that receives the signal, must be a reference type. If the expression evaluates to `null`, a run-time error occurs. The reference type must have a method with the above-mentioned name. The *Arguments* of the signal must be compatible with the parameter types of the method.

Notice that the evaluation order of a `send` statement is not left-to-right. The *Expression* is evaluated before the *Arguments*.

8.7 The `if` Statement

The `if` statement conditionally selects one of two code branches for execution.

IfThenStatement ::=
`if (` *Expression* `)` *Statement*

IfThenElseStatement ::=
`if (` *Expression* `)` *StatementNoShortIf* `else` *Statement*

IfThenElseStatementNoShortIf ::=
`if (` *Expression* `)` *StatementNoShortIf* `else` *StatementNoShortIf*

The contained expression must be of type `boolean`, or a compile-time error occurs. When executing the `if` statement, the expression is evaluated first. If the resulting value is `true`, the first contained statement is executed.

If the value is `false`, the second contained statement is executed. If there is no second statement (no `else` part), nothing is done.

The nonterminal symbols ending with *NoShortIf* are related to the infamous "dangling else" problem, which is demonstrated by the code fragment below.

```
if (cond1) if (cond2) x = 1; else x = 2;
```

Which `if` statement should the `else` branch be associated with? The choice in many languages, including Java, has been to associate an `else` with the innermost (rightmost) `if`, but this policy must somehow be enforced in the grammar. The solution here is to restrict the kind of statements allowed just before the keyword `else`. Such statements are explicitly forbidden to end with a "short if", i.e. an `if` statement without the `else` branch. This is the meaning of the *StatementNoShortIf* symbol. The result is that an `else` is always associated with the closest `if` statement possible. Because the statement before `else` could be any complex construct that has a "short if" as a trailing substatement, nonterminal symbols with *NoShortIf* at the end have been used in the grammar to represent statements that do not end with a "short if".

8.8 The `assert` Statement

An assertion is a statement that expresses an invariant that is checked at run-time.

AssertStatement ::=
    `assert` *Expression* `;`

The *Expression* must have type `boolean`, or a compile-time error occurs. When the assertion is executed, the *Expression* is evaluated, and it is a run-time error if the result is `false`.

This is a simplified version of assertions in Java, where an error message can be attached to an `assert` statement, and assertions may be disabled dynamically.

8.9 The `switch` Statement

The `switch` statement represents a choice with multiple possibilities.

SwitchStatement ::=
`switch (` *Expression* `) {` *SwitchGroup** `}`

SwitchGroup ::=
(`case` *Expression* | `default`) `:` *BlockStatement**

The expression in parentheses is known as the *switch expression*. The resulting type must be `int` or an enum type, or a compile-time error occurs. The *switch block*, enclosed in braces, may contain zero or more *case labels*, at most one *default label*, and zero or more statements. The mutual order of case and default labels is free. If the statements include local variable declarations, their scope is restricted to the switch block (unless tighter restrictions exist by other rules).

The types of expressions within case labels, known as *case expressions*, must be subtypes of the type of the switch expression, or a compile-time error will result. If the switch expression is an enum type, Java allows (or requires) the case expressions to be unqualified enumeration constant names. In Jumbala, the case expressions must be properly qualified as they are evaluated in the scope just like any other expression.

Java restricts case expressions to have constant values. Because Jumbala lacks the notion of constant expressions, this rule has been relaxed so that any expression of appropriate type is accepted as a case expression. As a result, the execution semantics of `switch` statements is slightly more general than in Java. To avoid confusion, the programmer is encouraged only to write case expressions that have constant, distinct, non-null values and no side effects. Under these assumptions the semantics in Java and Jumbala are equivalent.

A `switch` statement is executed by first evaluating the switch expression. If the value is `null`, execution terminates with a runtime error. Otherwise, each case expression is evaluated in turn and the resulting value is compared to the value of the switch expression. Case expressions are evaluated in the same order that they appear in the switch block. If there is a match (the values are equal), no further case expressions are evaluated, and execution jumps to the next statement after the case label. If no case matches and there is a default label, execution jumps to the statement following the default label. If there is no match and no default label, no action is taken after evaluating all the case expressions and the switch statement completes.

After execution has jumped to a statement, that statement and all following statements within the switch block are executed sequentially, ignoring all case and default labels between the statements. If execution reaches an unlabeled `break` statement that is not further enclosed in a `switch`, `while`, `do`, or `for` statement, execution of the switch statement completes and no further statements in the switch block are executed.

8.10 The `while` Statement

The `while` statement executes a statement continuously as long as a given condition is true.

WhileStatement ::=
`while (` *Expression* `)` *Statement*

WhileStatementNoShortIf ::=
`while (` *Expression* `)` *StatementNoShortIf*

It is a compile-time error if the *Expression* does not have type `boolean`.

A `while` statement is executed by evaluating the expression. If it is `true`, the contained statement is executed and the entire `while` statement is executed again. If an unlabeled `break` statement is encountered, execution jumps out of the iteration, and an unlabeled `continue` statement skips back to testing the condition.

More formally, the semantics can be reduced to those of the `for` statement. A general `while` statement of the form

`while (`*condition*`)`
  body

can always be rewritten as

`for (;` *condition*`; )`
  body

The execution semantics of Section 8.12 can now be applied, even in the presence of `break` and `continue` statements.

8.11 The `do` Statement

The `do` statement repeats a loop until a Boolean condition is false.

DoStatement ::=
`do` *Statement* `while (` *Expression* `) ;`

The contained expression must have type `boolean` or a compile-time error will result.

Informally, on each iteration the statement is executed first and then the expression is evaluated. If the result is true, the entire `do` statement is executed again. Thus, the statement is always executed at least once. If an unlabeled `break` is executed inside the body of the loop, the `do` statement completes immediately. An unlabeled `continue` statement jumps straight to testing the Boolean expression.

Again, an arbitrary `do` statement

```
do
  body
while (condition);
```

can be expressed in terms of the more general `for` statement:

```
for (boolean c = true; c; c = condition)
  body
```

The symbol c above represents a unique identifier that does not appear anywhere else in the program. The semantics of the `for` statement (Section 8.12) now applies.

8.12 The `for` Statement

The `for` statement is a flexible loop construct that first executes an initialization code and then iterates by evaluating a Boolean condition, a statement, and an update code as long as the condition is true.

ForStatement ::=
`for (` *ForInitialization* `;` *ForCondition* `;` *ForUpdate* `)`
Statement

ForStatementNoShortIf ::=
`for (` *ForInitialization* `;` *ForCondition* `;` *ForUpdate* `)`
StatementNoShortIf

ForInitialization ::=
    LocalVariableDeclaration
  | *StatementExpressionList*

ForCondition ::=
    [*Expression*]

ForUpdate ::=
    StatementExpressionList

StatementExpressionList ::=
    [*StatementExpression* (, *StatementExpression*)*]

The *Statement* or *StatementNoShortIf* is called the *body* of the `for` statement. If *ForCondition* contains an expression, it must have type `boolean`, or a compile-time error occurs.

The execution begins at the initialization code. If *ForInitialization* is empty, no action is taken. If it consists of statement expressions, those are evaluated from left to right and the results are discarded. If *ForInitialization* is a local variable declaration, it is executed in the same way as a local variable declaration statement (Section 8.2. The scope of the declared variables is the rest of the `for` statement, up to and including the body. After initialization, the loop iteration begins.

Iteration of the loop has several phases. First, *ForCondition* is evaluated. If the resulting value is `false`, no further action is taken and the entire `for` statement completes. If the value is `true` or if the condition is empty, the body is executed. Then the expressions in *ForUpdate*, if any, are evaluated from left to right and the results are discarded. Finally a new iteration step is taken.

If execution of the body reaches an unlabeled `break` statement that is not enclosed in the body of a nested `switch`, `while`, `do`, or `for` statement, the iteration stops immediately. *ForUpdate* or *ForCondition* are not evaluated, and execution of the entire `for` statement completes.

If an unlabeled `continue` statement is executed in the contained statement, but not within a nested `while`, `do`, or `for` statement, the rest of the contained statement is skipped. The expressions in *ForUpdate* are evaluated, and a new iteration step begins.

The current version of Java also defines so called enhanced `for` statements, which are a syntactical shortcut for iterator-based looping. This construct is not allowed in Jumbala.

8.13 The `break` Statement

The `break` statement transfers control out of an enclosing statement.

BreakStatement ::=
    `break` [IDENTIFIER] `;`

A `break` statement can be unlabeled or labeled with an identifier.

When executed, an unlabeled `break` statement terminates the execution of the innermost enclosing `switch`, `while`, `do`, or `for` statement, transferring control to the end of the statement. If an unlabeled `break` statement is not contained in such enclosing structure, a compile-time error will occur.

A labeled `break` statement must be enclosed in a *LabeledStatement* with the same label, or a compile-time error occurs. When the `break` statement is executed, control is immediately transferred to the end of the *LabeledStatement*. Unlike unlabeled `break` statements, labeled `break` statements are not restricted to be used only with iteration or `switch` statements.

8.14 The `continue` Statement

The `continue` statement transfers control to the loop continuation point of an iteration statement.

ContinueStatement ::=
    `continue` [IDENTIFIER] `;`

A `continue` statement can be unlabeled or labeled with an identifier.

The *continue target* of an unlabeled `continue` statement is the innermost `while`, `do`, or `for` statement enclosing the `continue` statement. The continue target of a `continue` statement labeled with an identifier is the `while`, `do`, or `for` statement that (i) encloses the `continue` statement and (ii) is immediately enclosed in a *LabeledStatement* labeled with the same identifier. In either case, if there is no continue target fulfilling the conditions, a compile-time error occurs.

If a `continue` statement is executed and the continue target is a `while` or `do` statement, execution will jump to the point of the iteration of the continue target where the Boolean condition is evaluated. Otherwise the continue target is a `for` statement, and execution will jump to the point of iteration where *ForUpdate* is evaluated.

8.15 The `return` Statement

The `return` statement is used to return from a method or constructor.

ReturnStatement ::=
`return` [*Expression*] `;`

It is a compile-time error if a `return` statement is not enclosed in a method or constructor body.

If a `return` statement appears in the body of a constructor or a `void` method, it must not contain an *Expression*, or a compile-time error occurs. When the statement is executed, control is transferred back to the level that invoked the constructor or method (Sections 9.15 and 9.14).

If a `return` statement appears in the body of a method that is not `void`, it must contain an *Expression* whose type is a subtype of the return type of the method, or a compile-time error occurs. When the statement is executed, the *Expression* is evaluated and the result is used as the return value of the method. Control is then transferred back to the level that invoked the method (Section 9.14).

Chapter 9

Expressions

Expressions are elements of a Jumbala program that denote computation of values. Expressions are evaluated at run time to produce values. Expressions can also have side effects, such as changing the value of a variable.

Expressions can appear in statements, in variable initializers, as case expressions in `switch` statements, and as subexpressions in other expressions. A common special case is an expression statement, which is a statement that contains just one expressions that is evaluated for side effects.

Expressions have a type that is deduced at compile-time. The type of the run-time value of an expression is a subtype (Section 4.5) of the type of the expression. An exception is a method invocation expression that invokes a method that returns no value (a `void` method). Such an expression has no type and produces no value, and it is a compile-time error if it appears anywhere else but as the expression in an expression statement.

We say that an expression produces an *lvalue* if the expression denotes a variable that can be assigned to. As a simple example, the expression `temp` is an lvalue if it appears in the scope of a local variable named `temp` that is not `final`. An lvalue can be used as a value, or it can be assigned to. Therefore, an expression producing an lvalue can appear as the left-hand side of an assignment or as the operand of an increment or decrement operator.

When expressions are assembled from simpler subexpressions, the evaluation order of subexpressions is fully specified in Jumbala, like in Java. Generally, the order is from left to right, and subexpressions are evaluated before the containing expression. Specifically, of the two operands of a binary operator, the leftmost one is evaluated first, then the rightmost one.

The set of different kinds of expressions is the same as in Java. We also define the grammar for expressions in a similar way. The grammar rules in this chapter define how complex expressions are divided syntactically. The hierarchy of grammar rules implies the rules for operator precedence and asso-

ciativity, which are the same as in Java. The nonterminal symbol *Expression* lies at the top of the hierarchy, representing any kind of an expression.

Expression ::=
    AssignmentExpression

9.1 Contexts of Expressions

An expression can appear in a *static context* or in an *instance context*. The context is an instance context if the expression appears inside one of the following constructs.

1. A variable initializer for an instance variable (Section 6.6).
2. The body of an instance method (Section 6.7).
3. The body of a constructor but not in an explicit superclass constructor invocation (Section 6.8).

Otherwise the context is a static context.

When an expression that appears in an instance context is evaluated, a *current object* is implicitly known. In cases 1 and 3 above, the current object is the object that is being created or initialized. In case 2, the current object is the object that the method has been invoked for, i.e. the object obtained by evaluating the target reference in the method invocation.

A `this` expression can be used in an instance context to obtain a reference to the current object. The word `this` may not appear in a static context.

9.2 Assignment Operators

As assignment expression assigns a value to a variable.

AssignmentExpression ::=
    TernaryConditionalExpression
  | *Assignment*

Assignment ::=
    LeftHandSide `=` *Expression*
  | *LeftHandSide CompoundAssignmentOperator Expression*

LeftHandSide ::=
VariableName | *FieldAccess* | *ArrayAccess*

CompoundAssignmentOperator ::=
`*=` | `/=` | `%=` | `+=` | `-=` | `<<=` | `>>=` | `>>>=` | `&=` | `^=` | `|=`

The left hand side of an assignment must be an lvalue, or a compile-time error occurs. In other words, `final` variables may not be assigned to. It is also a compile-time error if the left hand side is a variable name whose resolution fails.

The type of an assignment expression is the type of the left hand side expression. The value of an assignment is the assigned value. Assignment does not produce an lvalue.

9.2.1 Simple Assignment

A simple assignment has the form *LeftHandSide* `=` *Expression*. The type of *Expression* must be a subtype of the type of *LeftHandSide*, or a compile-time error occurs. Execution of a simple assignment can be divided to three disjoint cases.

If the left hand side is a simple variable name (a single identifier) or a qualified variable name qualified by a type name or a field access expression of the form `super.`IDENTIFIER, the right hand side *Expression* is evaluated and the result is assigned to the variable denoted by the left hand side.

If the left hand side is a field access expression of the form *PrimaryExpressionNoName* `.` IDENTIFIER or a qualified variable name qualified by another variable name, the qualifying expression or variable name is first evaluated to produce a value, call it *ref*. Then the right hand side is evaluated to produce a value. If the left hand side denotes a class variable, the result is assigned to the variable. Otherwise the left hand side denotes an instance variable. If *ref* is `null`, a run-time error occurs. Otherwise the result of the right hand side is assigned to the object referred to by *ref*.

If the left hand side is an array access expression, it has the form *refexpr*`[`*indexexpr*`]`. First, *refexpr* is evaluated to produce a value *ref*. Then, *indexexpr* is evaluated to produce a value *index*. Then, the right hand side is evaluated to produce a value *rhs*. If *ref* is `null`, a run-time error occurs at this point. Otherwise *ref* is a reference to an array. If *index* is equal to or greater than the length of the array or if *index* is less than zero, a run-time error occurs. If the component type of the array object is a reference type and if *rhs* is a reference to an object whose type is not a subtype of the component type, a run-time error occurs. This is situation is equivalent to the

ArrayStoreException in Java. Otherwise *rhs* is assigned to the component of the array whose index is *index*.

9.2.2 Compound Assignment

A compound assignment has the form *lhsexpr op*`=` *rhsexpr*, where *op* is one of the binary operators `*`, `/`, `%`, `+`, `-`, `<<`, `>>`, `>>>`, `&`, `^`, or `|`. It is roughly equivalent to the expression *LeftHandSide* `=` *LeftHandSide op Expression*.

The types of the left hand side and right hand side expressions must form a valid pair of operand types for the binary operator, or a compile-time error occurs. Because of the simplified type system of Jumbala, this restriction implies that the result type is assignable to the variable on the left hand side.

At run-time, the left hand side is evaluated to produce a value of a variable. We call the variable *var* and its value *lhs*. Both the identity of the variable and its value are stored at this point. Then the right hand side is evaluated to a value, call it *rhs*. The binary operator is applied to the values as if by evaluating *lhs op rhs*. The result is assigned to *var*. Like in Java, evaluation of the right hand side cannot not change the value *lhs* or the target variable of the assignment, because they have already been fixed before the right hand side is evaluated.

9.3 Conditional Operators

The three conditional operators perform conditional branching in expressions. Conditional operators do not produce lvalues.

9.3.1 The Ternary Conditional Operator

A ternary conditional operator can be seen as an inline replacement for an `if` statement.

TernaryConditionalExpression ::=
    ConditionalOrExpression
  | *ConditionalOrExpression* `?`
      Expression `:` *TernaryConditionalExpression*

The expression before the question mark is called the *condition* of the ternary conditional expression. The expressions before and after the colon are the *true-expression* and *false-expression*, respectively.

The condition must have type `boolean`, or a compile-time error occurs. The type of the ternary conditional expression depends on the types of the subexpressions. If the type of the true-expression is a subtype of the false-expression, the result type is the type of the false-expression. Otherwise, if the type of the false-expression is a subtype of the true-expression, the result type is the type of the true-expression. It is a compile-time error if neither holds, or if either or both of the subexpressions is invocation of a `void` method.

The type rules above are equivalent to those of the previous versions of Java. Java 5.0 allows any types of true-expressions and false-expressions as long as they have a common supertype. In Jumbala, the same can be achieved by explicitly casting the true-expression (or the false-expression) to the common supertype.

At run-time, the condition is first evaluated and a subexpression is chosen for evaluation. If the result is `true`, the true-expression is chosen. Otherwise the false-expression is chosen. The chosen expression is evaluated and its value becomes the value of the ternary conditional expression. The expression that is not chosen will not be evaluated.

9.3.2 The Other Conditional Operators

The conditional or operator and the conditional and operator perform logical operations with short-circuit evaluation. They work in the same way as in Java.

ConditionalOrExpression ::=
    ConditionalAndExpression
  | *ConditionalOrExpression* `||` *ConditionalAndExpression*

ConditionalAndExpression ::=
    OrExpression
  | *ConditionalAndExpression* `&&` *OrExpression*

A conditional or expression has the form *leftoperand* `||` `rightoperand` and it is equivalent to the ternary conditional expression `((`*leftoperand*`) ? true : (`*rightoperand*`))` with respect to compile-time restrictions and run-time evaluation.

Similarly, a conditional and expression *leftoperand* `&&` *rightoperand* is equivalent to `((`*leftoperand*`) ? (`*rightoperand*`) : false)`.

9.4 Bitwise and Logical Operators

The bitwise or, bitwise exclusive or, and bitwise and operators combine integer values to produce an integer result. The corresponding logical operators combine `boolean` values to produce `boolean` results. The semantics is equivalent to that in Java.

OrExpression ::=
    OrExpression
  | *OrExpression* `|` *XorExpression*

XorExpression ::=
    AndExpression
  | *XorExpression* `^` *AndExpression*

AndExpression ::=
    EqualityExpression
  | *AndExpression* `&` *EqualityExpression*

Bitwise and logical operators are not distinguished syntactically but on the grounds of the types of their operands. If both operands expressions have type `int`, the operator is a bitwise operator and the result type is `int`. If both operands have type `boolean`, the operator is a logical operator and the result type is `boolean`. If neither holds, a compile-time error occurs.

At run time, the left operand is evaluated first, then the right operand. Unlike the conditional operators, logical operators do not apply short-circuit evaluation rules. Both operands are always evaluated.

For integer operands, the `|` operator performs a bitwise inclusive or operation on the operand values. The `^` operator performs a bitwise exclusive or operation, and the `&` operator performs a bitwise and operation.

For `boolean` operands, the value of *a* `|` *b* is `true` if and only if *a* and *b* are not both `false`. The value of *a* `^` *b* is `true` if and only if one of *a* and *b* is `true` and the other is `false`. The value of *a* `&` *b* is `true` if and only if both *a* and `b` are `true`.

A bitwise or logical operator never produces an lvalue.

9.5 Equality Operators

An equality expression tests whether its operands have the same value.

EqualityExpression ::=
    RelationalExpression
  | *EqualityExpression* (`==` | `!=`) *RelationalExpression*

An equality expression is not an lvalue. The resulting type is `boolean`. The type of the left operand expression must be castable to the type of the right operand expression (or vice versa; castability is a symmetric relation). Otherwise a compile-time error occurs.

The operator `==` evaluates to `true` if and only the run-time values of the operands are equal. All such cases are listed below. The operator `!=` evaluates to `true` if and only if the operands are not equal.

Two values a and b are equal if and only if one of the following holds.

1. The value a is `true` and b is `true`.
2. The value a is `false` and b is `false`.
3. The value a is an `int` and b is an `int` and both represent the same integer.
4. The value a is `null` and b is `null`.
5. The value a is a reference to an object and b is a reference to the same object.

9.6 Relational Operators

The relational operators perform numerical comparison of integer values or type comparison of reference types.

RelationalExpression ::=
    ShiftExpression
  | *RelationalExpression* (`<` | `>` | `<=` | `>=`) *ShiftExpression*
  | *RelationalExpression* `instanceof` *ReferenceType*

The type of a relational expression is always `boolean`. A relational expression is never an lvalue.

The operators `<`, `>`, `<=`, and `>=` are the numerical comparison operators. They require that both operand expressions are of type `int`. Otherwise a compile-time error occurs. The value of a numerical comparison expression is `true` or `false`, depending on the signed integer values of the operands as follows. The value produced by operator `<` (operator `>`) is `true` if and only if the left operand is strictly less than (greater than) the right operand. The value produced by operator `<=` (operator `>=`) is `true` if and only if the left operand is less than or equal to (greater than or equal to) the right operand.

The type comparison operator `instanceof` has one operand appearing on the left-hand side. The type of the operand expression, call it T, must be a reference type or the null type. The *ReferenceType* appearing on the right-hand side must denote a type that is castable to T. Failure to meet these conditions produces a compile-time error. The run-time value of an `instanceof` expression is `true` if and only if the value of the operand is a reference to an object whose type is a subtype of the *ReferenceType*. If the operand is `null`, the `instanceof` expression evaluates to `false`.

The `instanceof` operator is closely related to cast expressions. Assume that `e` is an expression and `T` is a type. If the expression `e instanceof T` produces no compile-time error and evaluates to `true` at run time, then the cast expression `(T) e` will not produce a run-time error (as long as `e` still has the same value, of course). The converse does not hold in all cases. The cast `(String) null` is correct even though `null instanceof String` evaluates to false. Furthermore, the cast `(int) 17` contains no errors, but the phrase `17 instanceof int` is not even a valid expression.

9.7 Shift Operators

The left shift operators `<<`, signed right shift operator `>>`, and unsigned right shift operator `>>>` perform bitwise shifting of integer values.

ShiftExpression ::=
    AdditiveExpression
  | *ShiftExpression* (`<<` | `>>` | `>>>`) *AdditiveExpression*

The types of the left and right operand expressions for all these operators must be `int`, or a compile-time error occurs. The left operand is the value to be shifted, and the right operand is the shift distance.

The resulting type of a shift expression is also `int`, and it is not an lvalue.

At run time, the left operand is evaluated first, then the right operand. The value of `n << s` is `n` shifted left by `s` bit positions, interpreted as a 32-bit signed integer. This corresponds to multiplying by 2^s.

The value of `n >> s` or `n >>> s` is obtained by shifting `n` right by `s` bits. If `n` is non-negative, the result is the same regardless of which operator is used. If `n` is negative (the highest-order bit of the two's-complement representation is 1), the signed operator `>>` uses sign-extension. In other words, the `s` highest-order bits of `n >> s` are ones. The unsigned operator `>>>` uses zero-extension, so the `s` highest-order bits of `n >>> s` are zeros regardless of the sign of `n`.

Signed right-shifting divides the left operand by the value $2^{\texttt{s}}$, rounding towards minus infinity. In contrast, the binary division operator `/` always rounds towards zero.

In Java, the value of `n` *op* `s` is equal to the value of `n` *op* `(s & 31)` if `n` is an `int` and *op* is a shift operator. Jumbala deviates from this behavior if `s` is less than zero or greater than 31. If `s` is negative, a run-time error occurs immediately after its evaluation. If `s` is strictly greater than 31, then all the bits of the left operand are shifted off the 32-bit boundary. Therefore, assuming that `s` ≥ 32, the value of `n << s` or `n >>> s` is 0. The value of `n >> s` is 0 if `n` is non-negative, and -1 otherwise, because of sign-extension.

9.8 Additive Operators

The additive operators `+` and `-` are used for integer addition and subtraction. The `+` operator is also used for string concatenation.

AdditiveExpression ::=
    MultiplicativeExpression
  | *AdditiveExpression* (`+` | `-`) *MultiplicativeExpression*

The operand expressions for the binary `+` operator can be either both of type `String` or both of type `int`. The types of both operand expressions for the binary `-` operator must be `int`. It is a compile-time error if these restrictions are not met.

The operator `+` acting on two `String` operands is called the string concatenation operator. Note specifically that it is not sufficient if only one of the operand expressions has the type `String`. Java is much more liberal in this aspect: if one operand is a `String`, the other can be of any type, and it will be implicitly converted to string form by an appropriate mechanism. Jumbala does not support implicit conversion, but an explicit invocation of the static method `valueOf` in class `String` can be used to achieve the same effect.

The run-time value of a string concatenation expression is a reference to a `String` object that has the characters of the left operand immediately followed by the characters of the right operand. If either operand evaluates to `null` at run time, a run-time error occurs. Java would convert a `null` value to the text string "null" instead.

The operator `+` is called the addition operator when acting on integer operands. The resulting value is the sum of the operands, with only the 32 lowest-order bits of the two's complement representation taken into account.

Similarly, the `-` operator subtracts the right-hand operand value from the left operand value. The value of `x - y` is always the same as `x + (-y)`. Notice that if addition or subtraction overflows, the result is silently truncated without an error message.

An additive expression never produces an lvalue.

9.9 Multiplicative Operators

The binary multiplicative operators `*`, `/`, and `%` perform arithmetic on integer values.

MultiplicativeExpression ::=
    UnaryExpression
  | *MultiplicativeExpression* (`*` | `/` | `%`) *UnaryExpression*

It is a compile-time error if either operand expression of a multiplicative operator is not of type `int`. A multiplicative expression first evaluates the left operand, then the right operand. The expression has type `int`, and it is not an lvalue.

The multiplication operator `*` yields the product of its two operands. The result has the 32 lowest-order bits of the true mathematical value. If an overflow occurs, the result may have a wrong magnitude, and possibly the wrong sign. This does not produce a run-time error or any other signal to the user.

The division operator `/` gives the value of the left operand divided by the value of the right operand. The result is rounded to a whole number towards zero. It is a run-time error if the right operand evaluates to zero. The only situation in which an overflow occurs is when the left operand is equal to -2^{31} and the right operand is equal to -1. The result in this case is -2^{31} instead of the true value 2^{31}, which cannot be represented in 32-bit two's-complement form.

The modulo operator `%` produces the remainder from the integer division of the left operand value x by the right operand value y. If y equals 0, a run-time error occurs. Assume that y is not 0. If x is an integer multiple of y, the result value is zero. Otherwise the result value, call it m, is an integer that has the same sign as x. The absolute value of m is less than the absolute value of y, and $x - m$ is an integer multiple of y. In all cases, if `x` and `y` have values of type `int` and `y` is not equal to 0, the expression `x % y` gives the same result as `x - ((x / y) * y)`. This is true even in the special case in which the division expression `x / y` overflows.

9.10 Unary Operators

Unary operators take only one operand instead of two. The unary operators in Jumbala are the same as in Java.

UnaryExpression ::=
    UnaryExpressionNotPlusMinus
  | `+` *UnaryExpression*
  | `-` (*UnaryExpression* | INT_LITERAL_MUST_NEGATE)
  | *PreIncrement*
  | *PreDecrement*

UnaryExpressionNotPlusMinus ::=
    PostfixExpression
  | *Cast*
  | `~` *UnaryExpression*
  | `!` *UnaryExpression*

9.10.1 Numerical Unary Operators

The unary plus operator `+`, the unary minus operator `-`, and the bitwise complement operator `~` are the numerical unary operators. The type of these expressions is `int`. The operand expression must also be `int`, or a compile-time error occurs.

The value of a unary plus expression `+`*UnaryExpression* is the same as the value of its operand. The unary plus operator does not contribute to the semantics of a program, but it exists as a contrast to the unary minus operator.

The value of a unary minus expression `-`*UnaryExpression* is the arithmetic negation of the operand value. Because of the two's-complement representation, the arithmetic negation of -2^{31}, the most negative integer value, is the same value itself. A related special case is the expression `-2147483648`, which consists of the unary minus followed by the token INT_LITERAL_MUST_NEGATE. This is the only place in the grammar where the token may appear.

The value of a bitwise complement expression `~`*UnaryExpression* is the value of the operand with all bits complemented. Thus, `~x` equals `-x-1` for all integer values of `x`.

A numerical unary expression is not an lvalue.

9.10.2 The Logical Complement Operator

The logical complement operator `!` must have an operand expression of type `boolean`, or a compile-time error occurs. The logical complement expression has type `boolean`, and it is not an lvalue.

The value of the logical complement expression is `true` if the operand is `false`. The value is `false` if the operand is `true`.

9.10.3 Casts

A cast expression is used to verify an assumption about the type of the operand expression.

Cast ::=
    `(` *PrimitiveType* `)` *UnaryExpression*
  | `(` *ArrayType* `)` *UnaryExpression*
  | `(` *TypeName* `)` *UnaryExpressionNotPlusMinus*

The type in parentheses is said to be the target type of the cast expression. It is a compile-time error if the operand expression has a `void` result or if type of the operand expression is not castable (Section 4.6) to the target type.

The type of a cast expression is its target type. A cast expression is never an lvalue, even if the operand expression is.

If the operand expression is of a primitive type, it is necessarily the same as the target type, by the definition of castability. The value of such a cast expression at run time is the value of the operand expression.

If the type of the operand expression is not a primitive type, it can only be a reference type or the null type. Then the run-time value of the operand is either a reference to an object or a `null` reference. A run-time error occurs if the operand value is a reference to an object whose type is not a subtype of the target type. Note that this can never happen if the type of the operand expression is a subtype of the target type. If there is no run-time error, the value of the cast expression is the same as the value of the operand.

The semantics can be stated informally as follows. A compile-time check is made that it is at least possible that the run-time type of the operand is a subtype of the target type. At run time, if there is any doubt, it is checked that the actual type of the operand is indeed a subtype of the target type. Notice that a cast expression never converts an object into an object of another type, or a value of a primitive type into a value of another type.

The requirement of castability makes it illegal to write a cast of an expression of type S to a completely unrelated type T. To be more specific, unrelatedness here means that S and T could never have a common subtype (other than possibly the null type, which is a subtype of all reference types). For example, S and T could be classes, neither of which is a superclass of the other. If such a cast was permitted, it would always produce an error at run time, except in the special case that the operand evaluates to `null`. The cast expression would just degenerate into an obscure way of testing for a `null` value.

Although the Java specification [2] talks about 'casting conversions' instead of castability of types, the implied semantics and compile-time checks are the same.

9.10.4 Increment and Decrement Operators

The postfix increment and postfix decrement operators are the two unary postfix operators of Jumbala. They, and the prefix increment and prefix decrement operators, are used to adjust the value of an integer variable.

PostfixExpression ::=
    PrimaryExpression
  | *PostIncrement*
  | *PostDecrement*

PostIncrement ::=
    PostfixExpression `++`

PostDecrement ::=
    PostfixExpression `--`

PreIncrement ::=
    `++` *UnaryExpression*

PreDecrement ::=
    `--` *UnaryExpression*

Increment and decrement expressions can be used as statement expressions evaluated for their side effects, or as parts of more complex expressions. The type of a prefix or postfix increment or decrement expression is `int`. It is not an lvalue.

The operand expression must be an lvalue of type `int`, or a compile-time error occurs. At run time, the operand is evaluated to produce an integer value. The value 1 (in the case of a prefix or postfix increment operator `++`) or -1 (in the case of a prefix or postfix decrement operator `--`) is added to the integer, using the same rules for arithmetic as the binary `+` operator (Section 9.8). The new value is stored into the variable denoted by the lvalue.

In the case of a postfix increment or decrement operator (an operand followed by `++` or `--`), the resulting value is the the original unmodified value of the subexpression. In the case of a prefix increment or decrement operator (`++` or `--` followed by an operand), the resulting value is the new value of the variable.

9.11 Primary Expressions

Primary expressions are the basic building blocks that are used as elements in more complex expressions. These include literals and the `this` keyword, as well as field and array accesses, method invocations, and expressions in parentheses.

PrimaryExpression ::=
    PrimaryExpressionNoName
  | *VariableName*

PrimaryExpressionNoName ::=
    PrimaryExpressionNoNameNoNewArray
  | *ArrayCreation*

PrimaryExpressionNoNameNoNewArray ::=
    Literal
  | *FieldAccess*
  | *ArrayAccess*
  | *MethodInvocation*
  | *ClassInstanceCreation*
  | `this`
  | `(` *Expression* `)`

Literal ::=

INT_LITERAL
| STRING_LITERAL
| `false`
| `true`
| `null`

The type of an integer literal is `int`. The type of `false` or `true` is `boolean`. The type of `null` is the null type. The type of a string literal is the top-level type `String`. A literal is never an lvalue.

The expression `this`, when used in the body of an instance method, denotes a reference to the object for which the method was invoked, i.e. the current object (Section 9.1). If used in the body of a constructor or in the initializer of an instance variable, the value of `this` is a reference to the object being constructed. The keyword `this` may not be used in an other context, or a compile-time error occurs. The type of a `this` expression is the innermost class in which it occurs. A `this` expression is never an lvalue.

Parentheses can be used to control the order of evaluation of expressions. The type and value of a parenthesized expression are the same as those of the contained expression. A parenthesized expression is an lvalue if and only if the contained expression is an lvalue.

9.12 Field Access Expressions

A field access expression is used to access a member field of an object, using the value of an expression as a reference to the object. Another way to access fields is to use variable names (Section 5.4), which are technically not regarded as field access expressions.

FieldAccess ::=

PrimaryExpressionNoName `.` IDENTIFIER
| `super` `.` IDENTIFIER

In the first form, the type of the *PrimaryExpression* must be a reference type, and the type must have exactly one member field with the name denoted by the IDENTIFIER. Otherwise a compile-time error occurs. The type of the field access expression is the type of the selected field.

At run time the *PrimaryExpression* is evaluated. If the field is `static`, the value of the expression is discarded. The value of the field access expression is the value of the field.

If the field is not `static`, the value of the *PrimaryExpression* is checked. If the value is `null`, a run-time error occurs. If the value is not `null`, it is a reference to an object that has the selected field as an instance variable. The value of the field access expression is the value of that variable.

The second form may only appear in an instance context (Section 9.1). It is used to access fields of the superclass of the directly enclosing class. The superclass must have exactly one member field whose name is the IDENTIFIER, or a compile-time error occurs. The type of the field access expression is the type of the selected field. If the field is a class variable, the run-time value of the field access expression is the value of the variable. If the field is an instance variable, the value of the field access expression is the value of the instance variable in the current object (Section 9.1).

A field access expression results in an lvalue if and only if the selected field is not `final`.

9.13 Array Access Expressions

An array access expression is used to access the components of an array object.

ArrayAccess ::=
    PrimaryExpressionNoNameNoNewArray `[` *Expression* `]`
  | *VariableName* `[` *Expression* `]`

The *VariableName* or *PrimaryExpressionNoNameNoNewArray* appearing before the square brackets is called the array reference expression. Its type must be an array type, or a compile-time error occurs. The type of the array access expression is the component type of the array type.

The *Expression* inside square brackets is the index expression. It must be of type `int`, otherwise a compile-time error occurs.

Evaluation of an array access expression proceeds in the following order. The array reference expression is evaluated first, then the index expression. The value of the array reference is checked not to be `null`. If it is, a run-time error occurs. Otherwise the index value is compared to the length of the referred array. It is a run-time error if the index is equal to or greater than the length, or if the index is negative. Otherwise the index is used to select a component of the array, and the value of the array access expression is the value of that component.

The order of evaluation is slightly different if the array access appears on the left side of a simple assignment expression (Section 9.2.1).

An array access expression always gives an lvalue, even if the array reference is a `final` variable.

Notice that the grammar rule explicitly forbids writing an array creation expression before the square brackets in array access. For a justification, consider the expression `new int[3][4]`, which is supposed to create a 3-by-4 integer matrix. However, without the restriction, a parser could see it as `(new int[3])[4]`, which is an attempt to access the component at index 4 in a newly created array of 3 integers. Thus, the latter interpretation is ruled out. In fact, so is the former, because Jumbala does not allow multidimensional array creation expressions that specify the length of more than one dimension.

9.14 Method Invocation Expressions

A method invocation expression calls a method.

MethodInvocation ::=
    PrimaryExpressionNoName `.` IDENTIFIER *Arguments*
  | `super` `.` IDENTIFIER *Arguments*
  | *MethodName Arguments*

MethodName ::=
    [*AmbiguousName* `.`] IDENTIFIER

Arguments ::=
    `(` [*Expression* (`,` *Expression*)*] `)`

Every method invocation has a list of arguments, which are expressions inside parentheses. The grammar rules above show three kinds of *invocation forms*, depending on which symbols appear before the arguments. The invocation form affects the way a method invocation is processed.

At compile time, one method is chosen as the *prototype method* based on the invocation form and arguments. At run time a method is chosen as the *invocation method*. The invocation method may be the prototype method or some other method that has the same signature (Section 6.9.3). The concept of delaying the selection of the invocation method until run-time is a key aspect in object-oriented programming.

9.14.1 Compile-Time Processing

The first step is to select a reference type to search for methods that match the invocation. The type to search depends on the invocation form.

- The form is *PrimaryExpressionNoName* `.`IDENTIFIER. The type to search is the type of the primary expression. It is a compile-time error if the type is not a reference type.
- The form is `super.`IDENTIFIER. It is a compile-time error if the method invocation appears in a static context (Section 9.1) or if it is directly enclosed in the class `Object`. The type to search is the direct superclass of the directly enclosing class.
- The form is *MethodName*. The type to search is resolved by the rules in Section 5.3. It is a compile-time error if the type is not a reference type.

The next step is to locate methods that are *applicable*. Let n be the number of expressions in *Arguments* and let $A_1, \ldots, A_n$ be their types. A method M is applicable if all of the following apply.

1. M is a member of the type to search.
2. The name of M is the IDENTIFIER that appears in the method invocation right before the arguments.
3. M has exactly n parameters.
4. The type of the k:th parameter of M is a supertype of A_k, for all $k = 1, \ldots, n$.

It is a compile-time error if no method is applicable. Next, the *most specific* method is chosen as follows.

Let M and N be two methods with the same name and same number of parameters. Let $P_1, \ldots, P_n$ be the parameter types of M and $Q_1, \ldots, Q_n$ the parameter types of N. Method M is *more specific* than method N if and only if P_k is a subtype of Q_k for $k = 1, \ldots, n$.

An applicable method M is *maximally specific* if no applicable method other than M is more specific than M. If there is exactly one maximally specific method, that is the most specific method. Otherwise a compile-time error occurs. Notice that unlike in Java, two different maximally specific methods necessarily have a different signature because a type cannot have two methods with the same signature in Jumbala.

The most specific method is chosen as the prototype method. Further checks are made on the method.

- If the invocation form is `super.`IDENTIFIER and the prototype method is `abstract`, a compile-time error occurs.
- If the invocation form is *MethodName*, where the method name is simple, and the prototype method is an instance method, the invocation must appear in an instance context (Section 9.1) and the prototype method must be a member of the directly enclosing class. Otherwise a compile-time error occurs.
- If the invocation form is *MethodName*, where the method name is qualified by a type name (Section 5.3), and the prototype method is an instance method, a compile-time error occurs.

The type of the method invocation expression is the return type of the prototype method. If the prototype method does not return a value (it is declared with the keyword `void`), the result of the expression is `void` and the method invocation cannot appear as a subexpression of another expression.

A method invocation is never an lvalue.

9.14.2 Run-Time Processing

The run-time processing involves possibly evaluating an expression to obtain a *target reference* to an object, evaluating the arguments, selecting the invocation method based on the prototype method chosen at compile-time, and invoking the invocation method. First, the target reference is computed. The procedure depends on the invocation form.

- The form is *PrimaryExpressionNoName* `.`IDENTIFIER. The primary expression is evaluated first. The result of the primary expression is the target reference.
- The form is `super.`IDENTIFIER. The target reference is a reference to the current object.
- The form is *MethodName*, where the method name is an IDENTIFIER. The target reference is a reference to the current object.
- The form is *MethodName*, where the method name is qualified by a type name. The prototype method is necessarily a class method. There is no target reference.

- The form is *MethodName*, where the method name is qualified by a variable name. The variable name is evaluated as an expression (Section 5.4. The result is the target reference.

Next the argument expressions are evaluated from left to right.

If the prototype method is a class method or if the invocation form is `super.`IDENTIFIER, the prototype method is chosen as the invocation method. If there is a target reference, its value is ignored. It may or may not be `null`. Even though the target reference is not used, the expression that produces it is evaluated for side effects before evaluating the arguments.

If the prototype method is an instance method and the invocation form is not `super.`IDENTIFIER, there is necessarily a target reference. The target reference is checked after evaluating the arguments. If it is `null`, a run-time error occurs. Otherwise the target reference is a reference to an object whose class has an instance method with the same signature as the prototype method. That method is chosen as the invocation method.

The invocation method is invoked as follows. New variables are created for the parameters of the method. The values of argument expressions are assigned to the variables. The body of the method is then executed. If the method is an instance method, the target reference refers to an object that is used as the current object when executing the body of the method.

The execution of the body ends when a `return` statement is executed or when execution reaches the end of the body. All local variables created in the method are destroyed. If the `return` statement contains an expression, the value of the expression becomes the value of the method invocation expression.

If execution reaches the end of the method body and the method has a return type (the method is not `void`), a run-time error occurs. Such a situation does not exist in Java because a Java compiler performs a static analysis to ensure that all execution paths end in a proper `return` statement if the method is not `void`.

9.14.3 Examples

The following example illustrates the difference between various invocation modes and between class, instance, and `abstract` methods.

```
abstract class Base {
  abstract void abst();
  void inst() {}
  static void clas() {}
}

class Derived extends Base {
  Derived good = this;
  Derived bad = null;

  void abst() {}
  void inst() {}
  static void clas() {}

  void test() {
    abst();                // Invokes Derived.abst
    good.abst();           // Invokes Derived.abst
    bad.abst();            // Run-time error
    ((Base) this).abst();  // Invokes Derived.abst
    ((Base) good).abst();  // Invokes Derived.abst
    ((Base) bad).abst();   // Run-time error
    super.abst();          // Compile-time error
    Derived.abst();        // Compile-time error

    inst();                // Invokes Derived.inst
    good.inst();           // Invokes Derived.inst
    bad.inst();            // Run-time error
    ((Base) this).inst();  // Invokes Derived.inst
    ((Base) good).inst();  // Invokes Derived.inst
    ((Base) bad).inst();   // Run-time error
    super.inst();          // Invokes Base.inst
    Derived.inst();        // Compile-time error

    clas();                // Invokes Derived.clas
    good.clas();           // Invokes Derived.clas
    bad.clas();            // Invokes Derived.clas
    ((Base) this).clas();  // Invokes Base.clas
    ((Base) good).clas();  // Invokes Base.clas
    ((Base) bad).clas();   // Invokes Base.clas
    super.clas();          // Invokes Base.clas
    Derived.clas();        // Invokes Derived.clas
  }
}
```

9.15 Class Instance Creation Expressions

A class instance creation expression is used to create a new object, which is an instance of a predetermined class.

ClassInstanceCreation ::=
`new` *TypeName Arguments*

The *TypeName* is resolved to determine the class being instantiated. The name must denote a class type that is not `abstract`, or a compile-time error occurs. Enum instances cannot be created using class instance creation expressions.

The expressions in parentheses are used as arguments to a constructor. The number and compile-time types of the expressions are used to select a constructor of the class type, using the same algorithm as for method invocations. A compile-time error occurs if there is no unique most specific constructor that matches the arguments.

The type of a class instance creation expression is the class being instantiated, and its value is a reference to the new object. The expression is not an lvalue.

Every time a class instance creation expression is evaluated, the following steps are taken.

1. A new object of the class type is created. The object has a variable allocated for each instance variable declared in the class type or one of its superclasses. The variables are initialized to their default values (Section 4.7.2).

2. The constructor arguments are evaluated left to right.

3. The selected constructor is invoked. The argument values are assigned to parameter variables of the constructor.

4. If the constructor is for the class `Object`, this step is skipped. Otherwise the constructor begins with an explicit or implicit superclass constructor invocation (Section 6.8). The arguments in the invocation are evaluated left to right, and steps 3 to 6 are taken for the superclass and the selected superclass constructor.

5. Each instance variable initializer in the class is evaluated with the new object as the current object and the resulting value is assigned to the corresponding instance variable. The initializers are processed left to right.

6. The rest of the body for the constructor is evaluated with the new object as the current object.

9.16 Array Creation Expressions

An array creation expression is used to create a new instance of an array type.

ArrayCreation ::=
    `new` (*PrimitiveType* | *TypeName*) `[` *Expression* `]` (`[` `]`)*
  | `new` (*PrimitiveType* | *TypeName*) `[` `]` (`[` `]`)* *ArrayInitializer*

ArrayInitializer ::=
    `{` [*VariableInitializer* (`,` *VariableInitializer*)*] [`,`] `}`

VariableInitializer ::=
    Expression | *ArrayInitializer*

The *PrimitiveType* or *TypeName* denotes the element type for the array. The *Expression* surrounded by square brackets is the dimension expression. If the element type cannot be resolved or if the type of the dimension expression is not `int`, a compile-time error occurs.

The element type can be any non-array type, even an `abstract` class or interface type. The type of the new array object is an n-dimensional array of the element type, where n is the number of square bracket pairs appearing in the array creation expression, including the one that contains the dimension expression.

At run time the dimension expression is evaluated to obtain the length for the array. If the length is less than zero, a run-time error occurs. Otherwise a new array object with the specified type and length is created. The components of the array are initialized to their default values (Section 4.7.2).

The type of an array creation expression is the type of the new array object, and the value of the expression is a reference to the array. It is not an lvalue.

The Java language allows an additional construct that make it more convenient to create multidimensional arrays. Jumbala requires the programmer to be more explicit to achieve the same effect, as demonstrated by the following examples.

```
String[][] calendar;
```

```
// Not allowed:  calendar = new String[12][31];
// Instead, an explicit loop is required.
calendar = new String[12][];
for (int i = 0; i < calendar.length; i++)
  calendar[i] = new String[31];
```

Bibliography

[1] Jori Dubrovin. Jumbala — an action language for UML state machines. Research Report A101, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, March 2006.

[2] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, third edition, 2005.

Appendix A

List of Differences between Jumbala and Java

The following list contains items that describe how Jumbala differs from version 5.0 of the Java language.

- No exceptions and hence no `throws` clauses and no `throw`, `try-catch`, or `try-catch-finally` statements.
- The following situations cause a run-time error instead of an exception: division by zero, following a null reference, an illegal cast, an illegal array store, creating an array with negative length, accessing an array out of bounds, trying to clone a non-`Cloneable` object, assertion error.
- No compile-time checking against unreachable statements or against reading uninitialized local variables. The latter causes a run-time error.
- No threading, hence no `synchronized` methods or `synchronized` statements, no `volatile` variables and no complicated memory model.
- No serialization support, no `transient` variables.
- No floating-point types `double` or `float`, no floating-point arithmetic or FP-strict expressions, classes, or interfaces. No floating-point literals.
- No instance initializers or static initializers.
- Strict left-to-right initialization order for fields with no compile-time checks against accessing an uninitialized field. No special treatment of final class variables with a constant initializer.

- All final variables must have an initializer. Assignment to final variables is never allowed in constructors or methods.
- No alternate constructor invocations.
- No recovering from out-of-memory situations.
- More intuitive overflowing rules for shift expressions, involving a run-time error when the shift distance is negative.
- No unused keywords such as `goto`.
- No compile-time checking against a missing return statement. Falling off the end of the body of a non-void method causes a run-time error.
- No implicit conversion of a type to `String`.
- No boxed types `Integer` or `Boolean`, hence no automatic boxing or unboxing.
- The only integer type is `int`. No `byte`, `short`, `char`, or `long` types. Hence no conversions between primitive types. No character literals or long integer literals.
- No dynamic class loading, hence no possibility for missing methods or other binary incompatibilities.
- No packages, no `import` declarations.
- Limited character set in identifiers. No support for Unicode in programs. No support for carriage returns as line terminators.
- No support for obsolete placing of array brackets, such as in `int a[];`.
- No inner classes, no local classes, no anonymous classes. All nested classes are static. The keyword `static` may not be used for declaring a nested class.
- No generic types, no type variables.
- Simpler definitions for built-in classes `Object` and `String`.
- No access control. Every declaration is implicitly public. No keywords `private`, `protected`, or `public`.
- The body of an enum may only contain enum constants without arguments or class bodies. No implicit valueOf method for enums.

- A declaration of a supertype must precede the declaration of a subtype.
- No annotations.
- No implicit initialization of multidimensional arrays.
- No `assert` statements that have an associated message string.
- No restriction that the `case` expressions of a `switch` statement must be compile-time constants. No checking against several `case` expressions evaluating to the same value. No special scoping rules for enum constants appearing in `case` expressions.
- No enhanced `for` statements.
- Added `send` statements.
- Added support for top-level statements and incremental programs.

Appendix B

Grammar Rules

All grammar rules used in this specification are listed below. Consult Section 1.1 for a description of the notation. The start symbol of the grammar is *Program*.

Program ::=
    $\big($*TypeDeclaration* | *BlockStatement*$\big)^*$

Type ::=
    PrimitiveType | *ReferenceType*

PrimitiveType ::=
    `int` | `boolean`

ReferenceType ::=
    ArrayType | *TypeName*

ArrayType ::=
    Type `[]`

TypeName ::=
    $\big[$*TypeName* `.` $\big]$ IDENTIFIER

MethodName ::=
    $\big[$*AmbiguousName* `.` $\big]$ IDENTIFIER

VariableName ::=
    $\big[$*AmbiguousName* `.` $\big]$ IDENTIFIER

AmbiguousName ::=
    [*AmbiguousName* `.`] IDENTIFIER

TypeDeclaration ::=
    ClassDeclaration
  | *InterfaceDeclaration*
  | *EnumDeclaration*

ClassDeclaration ::=
    [`abstract` | `final`] `class` IDENTIFIER
        [`extends` *TypeName*]
        [`implements` *TypeName* (`,` *TypeName*)*]
        `{` *ClassBodyDeclaration** `}`

ClassBodyDeclaration ::=
    MemberDeclaration
  | *ConstructorDeclaration*

MemberDeclaration ::=
    FieldDeclaration
  | *MethodDeclaration*
  | *TypeDeclaration*

InterfaceDeclaration ::=
    `interface` IDENTIFIER
        [`extends` *TypeName* (`,` *TypeName*)*]
        `{` *MemberDeclaration** `}`

EnumDeclaration ::=
    `enum` IDENTIFIER `{` [IDENTIFIER (`,` IDENTIFIER)*] [`,`] `}`

FieldDeclaration ::=
    (`static` | `final`)* *Type*
        VariableDeclarator (`,` *VariableDeclarator*)* `;`

VariableDeclarator ::=
    IDENTIFIER [`=` *VariableInitializer*]

VariableInitializer ::=
    Expression | *ArrayInitializer*

ArrayInitializer ::=
    `{` [*VariableInitializer* (`,` *VariableInitializer*)*] [`,`] `}`

MethodDeclaration ::=
    (`abstract` | `static` | `final` | `native`)* (*Type* | `void`)
        IDENTIFIER *FormalParameters* *MethodBody*

FormalParameters ::=
    `(` [[`final`] *Type* IDENTIFIER (`,` [`final`] *Type* IDENTIFIER)*] `)`

MethodBody ::=
    `{` *BlockStatement** `}`
  | `;`

ConstructorDeclaration ::=
    IDENTIFIER *FormalParameters* *ConstructorBody*

ConstructorBody ::=
    `{` [`super` *Arguments* `;`] *BlockStatement** `}`

BlockStatement ::=
    LocalVariableDeclarationStatement
  | *Statement*

Statement ::=
    StatementWithoutTrailingSubstatement
  | *LabeledStatement*
  | *IfThenStatement*
  | *IfThenElseStatement*
  | *WhileStatement*
  | *ForStatement*

StatementNoShortIf ::=
    StatementWithoutTrailingSubstatement
  | *LabeledStatementNoShortIf*
  | *IfThenElseStatementNoShortIf*
  | *WhileStatementNoShortIf*
  | *ForStatementNoShortIf*

StatementWithoutTrailingSubstatement ::=
    Block
  | *EmptyStatement*
  | *ExpressionStatement*
  | *SendStatement*
  | *AssertStatement*
  | *SwitchStatement*
  | *DoStatement*
  | *BreakStatement*
  | *ContinueStatement*
  | *ReturnStatement*

Block ::=
    `{` *BlockStatement** `}`

LocalVariableDeclarationStatement ::=
    LocalVariableDeclaration `;`

LocalVariableDeclaration ::=
    [`final`] *Type VariableDeclarator* (`,` *VariableDeclarator*)*

LabeledStatement ::=
    IDENTIFIER `:` *Statement*

LabeledStatementNoShortIf ::=
    IDENTIFIER `:` *StatementNoShortIf*

EmptyStatement ::=
    `;`

ExpressionStatement ::=
    StatementExpression `;`

StatementExpression ::=
    Assignment
  | *PreIncrement*
  | *PreDecrement*
  | *PostIncrement*
  | *PostDecrement*
  | *MethodInvocation*
  | *ClassInstanceCreation*

SendStatement ::=
`send` IDENTIFIER *Arguments* `to` *Expression* `;`

Arguments ::=
`(` $\left[\textit{Expression}\ \left(\texttt{,}\ \textit{Expression}\right)^{*}\right]$ `)`

IfThenStatement ::=
`if (` *Expression* `)` *Statement*

IfThenElseStatement ::=
`if (` *Expression* `)` *StatementNoShortIf* `else` *Statement*

IfThenElseStatementNoShortIf ::=
`if (` *Expression* `)` *StatementNoShortIf* `else` *StatementNoShortIf*

AssertStatement ::=
`assert` *Expression* `;`

SwitchStatement ::=
`switch (` *Expression* `) {` *SwitchGroup** `}`

SwitchGroup ::=
$\left(\texttt{case}\ \textit{Expression} \mid \texttt{default}\right)$ `:` *BlockStatement**

WhileStatement ::=
`while (` *Expression* `)` *Statement*

WhileStatementNoShortIf ::=
`while (` *Expression* `)` *StatementNoShortIf*

DoStatement ::=
`do` *Statement* `while (` *Expression* `) ;`

ForStatement ::=
`for (` *ForInitialization* `;` *ForCondition* `;` *ForUpdate* `)`
Statement

ForStatementNoShortIf ::=
`for (` *ForInitialization* `;` *ForCondition* `;` *ForUpdate* `)`
StatementNoShortIf

ForInitialization ::=
    LocalVariableDeclaration
  | *StatementExpressionList*

ForCondition ::=
    [*Expression*]

ForUpdate ::=
    StatementExpressionList

StatementExpressionList ::=
    [*StatementExpression* (, *StatementExpression*)*]

BreakStatement ::=
    `break` [IDENTIFIER] `;`

ContinueStatement ::=
    `continue` [IDENTIFIER] `;`

ReturnStatement ::=
    `return` [*Expression*] `;`

Expression ::=
    AssignmentExpression

AssignmentExpression ::=
    TernaryConditionalExpression
  | *Assignment*

Assignment ::=
    LeftHandSide `=` *Expression*
  | *LeftHandSide CompoundAssignmentOperator Expression*

LeftHandSide ::=
    VariableName | *FieldAccess* | *ArrayAccess*

CompoundAssignmentOperator ::=
    `*=` | `/=` | `%=` | `+=` | `-=` | `<<=` | `>>=` | `>>>=` | `&=` | `^=` | `|=`

TernaryConditionalExpression ::=
    ConditionalOrExpression
  | *ConditionalOrExpression* `?`
      Expression `:` *TernaryConditionalExpression*

ConditionalOrExpression ::=
    ConditionalAndExpression
  | *ConditionalOrExpression* `||` *ConditionalAndExpression*

ConditionalAndExpression ::=
    OrExpression
  | *ConditionalAndExpression* `&&` *OrExpression*

OrExpression ::=
    OrExpression
  | *OrExpression* `|` *XorExpression*

XorExpression ::=
    AndExpression
  | *XorExpression* `^` *AndExpression*

AndExpression ::=
    EqualityExpression
  | *AndExpression* `&` *EqualityExpression*

EqualityExpression ::=
    RelationalExpression
  | *EqualityExpression* (`==` | `!=`) *RelationalExpression*

RelationalExpression ::=
    ShiftExpression
  | *RelationalExpression* (`<` | `>` | `<=` | `>=`) *ShiftExpression*
  | *RelationalExpression* `instanceof` *ReferenceType*

ShiftExpression ::=
    AdditiveExpression
  | *ShiftExpression* (`<<` | `>>` | `>>>`) *AdditiveExpression*

AdditiveExpression ::=
    MultiplicativeExpression
  | *AdditiveExpression* (`+` | `-`) *MultiplicativeExpression*

MultiplicativeExpression ::=
    UnaryExpression
  | *MultiplicativeExpression* (`*` | `/` | `%`) *UnaryExpression*

UnaryExpression ::=
    UnaryExpressionNotPlusMinus
  | `+` *UnaryExpression*
  | `-` (*UnaryExpression* | INT_LITERAL_MUST_NEGATE)
  | *PreIncrement*
  | *PreDecrement*

UnaryExpressionNotPlusMinus ::=
    PostfixExpression
  | *Cast*
  | `~` *UnaryExpression*
  | `!` *UnaryExpression*

Cast ::=
    `(` *PrimitiveType* `)` *UnaryExpression*
  | `(` *ArrayType* `)` *UnaryExpression*
  | `(` *TypeName* `)` *UnaryExpressionNotPlusMinus*

PostfixExpression ::=
    PrimaryExpression
  | *PostIncrement*
  | *PostDecrement*

PostIncrement ::=
    PostfixExpression `++`

PostDecrement ::=
    PostfixExpression `--`

PreIncrement ::=
    `++` *UnaryExpression*

PreDecrement ::=
    `--` *UnaryExpression*

PrimaryExpression ::=
    PrimaryExpressionNoName
  | *VariableName*

PrimaryExpressionNoName ::=
    PrimaryExpressionNoNameNoNewArray
  | *ArrayCreation*

PrimaryExpressionNoNameNoNewArray ::=
    Literal
  | *FieldAccess*
  | *ArrayAccess*
  | *MethodInvocation*
  | *ClassInstanceCreation*
  | `this`
  | `(` *Expression* `)`

Literal ::=
    INT_LITERAL
  | STRING_LITERAL
  | `false`
  | `true`
  | `null`

FieldAccess ::=
    PrimaryExpressionNoName `.` IDENTIFIER
  | `super` `.` IDENTIFIER

ArrayAccess ::=
    PrimaryExpressionNoNameNoNewArray `[` *Expression* `]`
  | *VariableName* `[` *Expression* `]`

MethodInvocation ::=
    PrimaryExpressionNoName `.` IDENTIFIER *Arguments*
  | `super` `.` IDENTIFIER *Arguments*
  | *MethodName Arguments*

ClassInstanceCreation ::=
`new` *TypeName Arguments*

ArrayCreation ::=
`new` (*PrimitiveType* | *TypeName*) `[` *Expression* `]` (`[` `]`)*
| `new` (*PrimitiveType* | *TypeName*) `[` `]` (`[` `]`)* *ArrayInitializer*