# Contents

# 1 Overview

Java$^{--}$ is an object-oriented language: programs in this language are defined solely in terms of classes. A class definition forms the basis for declaration and creation of objects. A class contains two kinds of members: *fields* and *methods*. Fields are data variables used to associate states with objects, whereas methods are collections of statements that operate on fields and other objects to manipulate the state and perform other computations.

This document will describe the essential elements of Java$^{--}$ and specify semantics of various constructs. It also describes the various semantic constraints that your semantic analyzer must enforce on a Java$^{--}$ program.

# 2 Notation, terminology, and vocabulary

The syntax is described in extended Backus-Naur form (EBNF).

## 2.1 Identifiers and names

Identifier and names refer to entities such as classes, instance variables, methods, parameters, local variables, and constants declared in a Java$^{--}$ program. There are two forms of names: simple names and qualified names. A simple name is a single identifier, whereas a qualified name consists of a name, a "." token, and an identifier.

```
Name ::=  SimpleName | QualifiedName
SimpleName ::= id
QualifiedName ::=  Name '.' id
```

Names are used in the following situations:

- In declarations (see Section 5), where an identifier may specify the name by which the declared entity will be known,
- in field access expressions (see Section 7.3.4), after the "." token, and
- in some method invocation expressions(see Section 7.3.5), after the "." token and before the "(" token.

Below we show different ways in which names can be used:
Examples:

```
class ClassName {
 int classInstance;
 void func (int parameter) {
    classInstance = parameter;
 }
}
class UseClassName {
 ClassName cNameInstance;
 void locfunc (ClassName instParam) {
     cNameInstance = instParam;
 }
}
...
ClassName anInstance = new ClassName();         // create an instance of ClassName
UseClassName uInstance = new UseClassName();    // create an instance of UseClassName
uInstance.cNameInstance = anInstance;           // object assignment
uInstance.cNameInstance.func(2);                // Method invocation
uInstance.cNameInstance.classInstance = 4;      // nested field access
...
```

In this example, the names are used for the following purposes:

| Identifier | Usage of Identifier |
|---|---|
| `ClassName`, `UseClassName` | Definitions of classes |
| `classInstance`, `cNameInstance` | Definitions of instance variables |
| `parameter`, `InstParam` | Definitions of formal parameters |
| `func`, `locfunc` | Definition of instance methods |
| `anInstance`, `uInstance` | Definitions of instances |
| `uInstance.cNameInstance` | Field access |
| `uInstance.cNameInstance.func` | Method access |
| `uInstance.cNameInstance.classInstance` | Field access |

Note that `uInstance.cNameInstance.func` is a field access expression. It has two components: `uInstance.cNameInstance` and `func`. Expression `uInstance.cNameInstance` denotes a primary expression (see Section 7.3) identifying a reference type, whereas `func` denotes a member method associated with the reference type.

# 3 Program

A user defines a Java$^{--}$ program by enumerating a set of class declarations(Section 5.1) in a file.

```
Program ::=  ClassDeclaration*
```

Each file contains a main class and a set of auxiliary classes. The main class is similar to other classes, except that it contains a special method, called `main`. We show an example Java$^{--}$ program below:

Examples:

```
// example program
  class Fib {   // main class
     void main (int lo) {
        int hi;

        hi = 1;
        while (hi < 50) {
          hi = lo + hi;
          lo = hi - lo;
        }
     }
  }
  class Foo { // auxiliary class
    int val;
    Foo() { val = 0;}
  }
```

In this program, `Fib` is the main class, whereas `Foo` is an auxiliary class. In order to run this program, you will need to do the following:

```
% j-- -o fib Fib.jm
% fib 4
```

In this step, we assume that `j--` is a compiler for Java$^{--}$ and it produces an executable `fib`. In reality, creation of an executable may involve assembling the code generated by your compiler, and linking it with other object files and libraries. Execution of `fib` will result in invoking the main method associated with `Fib`.

**Compile Time Error:** Flag an error if there are more than one main classes or if there is no main class in a file.

# 4 Data types

A data type is defined by a set of values and operations over that set. Thus, the data type of a variable determines both the values that a variable may assume and the valid operations that can be performed on that variable.

Java$^{--}$ specifies two kinds of types: primitive type and reference type.

```
Type ::= PrimitiveType | ReferenceType
```

## 4.1 Primitive types

There are four kinds of primitive types defined: `int`, `boolean`, `char`, and `string`.

```
PrimitiveType ::=  int | boolean | char | string
```

The values of type `int` are a subset of the integers whose exact range is implementation dependent. The values of type `boolean` are `true` and `false` denoted by the reserved identifiers `true` and `false`. The values of `char` are defined by the ASCII character set.

The `string` data type is used to represent string objects. It provides the following methods for manipulating string objects:

```
string();
 // constructs a new string with value ""
string (string val);
 // constructs a new string that is a copy of val.
int index(char c, int start);
 // returns index of character 'c' in string with first position of c >= start
 // if c is not found, returns -1
 // if start > length(this) or if start < 0, returns -2
string +(string in1, string in2);
 // returns a string that contains characters in string in1 followed
 // by characters in string in2.
```

## 4.2 Reference type (class type)

Complex data structures in Java$^{--}$ are defined through reference types. Unlike Java, Java$^{--}$ has only one kind of reference types — classes. Class types are used to denote a set of objects. A class contains methods (collection of executable code) and data structures that capture the state of an instance of the class.

```
ReferenceType ::= ClassType
ClassType ::=  SimpleName
```

Unlike most programming languages (such as C++, Pascal, etc.) where complex data structures are usually allocated on the stack, the default mechanism in Java$^{--}$ is to create a reference to an instance of a class. For instance, the following declaration

```
RefType x;
```

is used to define a reference, called `x`, to an instance of `RefType`. `x` is initialized to `null`. An instance of `RefType` is explicitly created through a new expression (see Sections 7.3.3).

```
x = new RefType();
```

An execution of the above statement results in creating an object of type `RefType` on the heap. Variable `x` refers to the newly created object.

# 5 Declarations

A declaration introduces an identifier for a class, variable, or method, and associates a set of semantic attributes (such as type, value and signature) with the identifier. Your compiler will analyze the declarations and enter them, along with their attributes and scopes, in corresponding symbol tables. The symbol table package that you designed and implemented in part 1 of the compiler project will be useful for capturing declarations in a program.

## 5.1 Class declarations

A class declaration introduces a new reference type and describes how the reference type is implemented:

```
ClassDeclaration ::=  class id [Extends] ClassBody
Extends ::=  extends ClassType
```

Note that definitions such as the following are not valid:

```
class A extends B.C.D.E.F
```

**Compile Time Error:** Flag an error if you come across such definitions.

A class body may contain declarations of fields and methods.

```
ClassBody ::=  '{' ClassBodyDeclaration* '}'
ClassBodyDeclaration ::=  ClassMemberDeclaration
ClassMemberDeclaration ::=  FieldDeclaration | MethodDeclaration
```

Note that, unlike C++, there is no ';' character following class declarations.

### 5.1.1 Field declarations

A field declaration consists of an identifier denoting the new variable and its type:

```
FieldDeclaration ::=  Type VariableDeclarators ';'
VariableDeclarators ::=  VariableDeclarator (',' VariableDeclarator)*
VariableDeclarator ::=  id
```

A variable is denoted by its identifier.

### 5.1.2 Method declarations

Method declarations serve to define parts of programs and to associate identifiers with them so that they can be activated by method invocation expressions (see Section 7.3.5).

```
MethodDeclaration ::=  MethodHeader MethodBody
MethodHeader ::=  Type MethodDeclarator | void MethodDeclarator
MethodDeclarator ::=  id '(' [FormalParameterList] ')'
FormalParameterList ::=  FormalParameter (',' FormalParameter)*
FormalParameter ::=  Type id
MethodBody ::=  '{' LocalVariableDeclarationStatement* Statement* '}'
                | ';'
```

The `MethodHeader` specifies the identifier naming the method, the types and names of the any formal parameter identifiers, and the type of the value that the method returns, if any. If there is no return value, the keyword `void` must be used in place of the return type. The signature of a method is defined by the types of its formal parameters.

Examples:

```
// example class
class Point {
  int x, y;                // field definitions
  Point(int x, int y) {    // constructor
    this.x = x;            // initialize instance variable
    this.y = y;            // use this to access instance variables
  }
  Point Add(int x,int y){  // Add method with two parameters
    Point retVal;          // local variable
    retVal = new Point(x, y); // create new Point Object
    retVal.x = reVal.x+this.x;
    retVal.y = retVal.y+this.y;
```

```
        return retVal;
      }
      ...                        // other functions
    }
```

A class may not define multiple methods with the same name. In other words, Java$^{--}$ does not support method name overloading. It is a compile-time error to define methods with identical names in a class. In addition, both variables and methods cannot have identical names. For instance, the following is not a legal definition:

```
class Foo {
    int ABC;
    int ABC() { return 0; }
};
```

It is a compile-time error to have more than one formal parameter with the same name. The parameters are passed by value. The scope of a formal parameter is the entire body of the method. It must not, therefore, be redeclared within the body of the method.

The statement sequence of the method body specifies the actions to be executed upon invocation of the method via a method invocation expression (see Section 7.3.5). The use of the method identifier within its declaration implies recursion.

Examples:

```
class PassByValue {
   void main(char flag) {
      int one = 1;
      DoubleIt(one);
   }
   void DoubleIt(int arg) {
      arg = arg * 2;
   }
}
```

Note that all declarations within a method must occur before any executable statements. Declaration statements declare a variable. Local variables exist only as long as the method containing the variable is executing. A local variable must be declared and initialized (See section 5.1.1) before it is referenced in either a statement or an expression. Otherwise, a compile-time error occurs.

Examples:

```
void f(int a)
```

```
{
  int x;
  int z;

  x = x + 5; // valid use of x
  x = z;     // error: use of z before initialization
  y = x + 2; // error: use of y before declaration
}
```

Note: If a method does not return a value, this must be indicated by returning `void`.

### 5.1.3 Scope

Whenever an identifier is created, it lives in a particular *namespace*. Identifiers in the same namespace must have unique names. All identifiers introduced in the formal parameter list of a method heading and in the declaration part of the associated scope are *local* to the method declaration. The scope of these identifiers encompasses the formal parameter list, declaration list, and statement list within the method. Variables are not known outside their scopes. In the case of local variables, their values are undefined at the beginning of the statement list. All identifiers used in a scope must be declared. An identifier may be declared in the declaration part and formal parameter list of a procedure declaration only once.

When an identifier is used to denote a class, method, or variable, the meaning of the name is determined by recursively searching in the following order:

1. Local variables declared in the current scope.
2. A member of the enclosing class.
3. A field member of other classes.

Unlike Java, identifiers in Java$^{--}$ must be unique within a scope, i.e., within a symbol table in the current scope. For instance, your program should not allow redefinition of a variable within a nested block:

```
int i;

i = 0;
while (i < 10) {
{
    int i; // redefinition of i: mark error
```

```
    i = i+1;
}
```

It is important to note that the scope of an instance or data member of a class includes all member methods. It is, therefore, possible that a method may access a member that has not yet been defined. This means that the semantic analyzer must first gather all members of a class before it can semantically analyze of the bodies of methods.

Examples:

```
class aClass {
   void func() {
      classVar = classVar + 1; // valid access to an instance variable.
  }
   int classVar;
}
```

### 5.1.4 Inheritance

Java$^{--}$ supports single inheritance that is used to support extensibility of classes. The optional `extends` clause (see Section 5.1) specifies the direct superclass of the current class. The direct superclass is the class from whose implementation the implementation of the current class is derived. A derived class inherits all instance variables and methods from its superclass.

You should use the following rules for variable and method declarations that are overridden in a subclass:

- Fields can not be overridden; they can only be hidden. If a class declares a field that has the same name as the field defined in its superclass, then the field in the superclass exists. However, it can be accessed only through the keyword `super` or another reference to the superclass's type, i.e., AnInstance.Fieldname.
- Definition of a method with a signature that is identical to the one defined in the superclass overrides the later method. The superclass method can be accessed through the keyword `super` or by explicitly referencing the superclass's type.

When a method is invoked on an object, the actual type of the object determines which implementation is used. Examples:

```
class Point {
    int x, y
    int getX();
```

```
    int getY();
}
class ColoredPoint extends Point {
   int color;
}
class Colored3DPoint extends ColoredPoint {int z;}
```

In this example, `Color3DPoint` extends `ColoredPoint`, which extends `Point`. An instance of `Color3DPoint` contains four instance variables: `x`, `y`, `color`, and `z`.

# 6 SimpleBlock

A block denotes a unit of execution. A set of statements are associated with each block:

```
SimpleBlock ::=  '{' Statement* '}'
```

Note that all references to variables within a block must have been defined in the current scope. Otherwise, a compile-time error occurs.

## 6.1 Statements

Statements denote executable instructions.

```
Statement ::= IfThenStatement
        | IfThenElseStatement
        | WhileStatement
        | SimpleBlock
        | EmptyStatement
        | ExpressionStatement
        | ContinueStatement
        | ReturnStatement
        | IOStatement
```

### 6.1.1 If statements

The `IfThenStatement` and `IfThenElseStatement` specify that a statement be executed only if a certain condition (Boolean expression) is true.

```
IfThenStatement ::=  if  '(' Expression ')' Statement
IfThenElseStatement ::=  if '(' Expression ')' Statement else Statement
```

Examples:

```
if (i < 0) i = 1 else i = 2
if (i < 0) i = -i
```

The expression between the delimiters `if` and `then` must be of type `boolean`. Note that an `else` clause is bound to the most recent `if` clause that does not have one.

### 6.1.2 WhileStatement

The `WhileStatement` is used to represent a conditional loop. The expression after the delimiters `while` must be of type `boolean`.

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

The semantic of while loop specifies that the (boolean) expression is evaluated first, and if it is true, the statement is executed repeatedly until the expression evaluates to false. A typical `while` loop looks like the following:

```
while (i < 10) {
     j = j + i;
     i = i + 1;
 }
```

### 6.1.3 Empty statement

The empty statement executes no action.

```
EmptyStatement ::=  ';'
```

### 6.1.4 Continue statement

A continue statement skips to the end of a loop's body and evaluates the expression that controls the loop. A continue is often used to skip over an element of a loop range that can be ignored or treated with trivial code.

```
ContinueStatement ::=  continue ';'
```

A `continue` statement must occur within a loop; otherwise a compile-time error occurs.

### 6.1.5 Return statement

A return statement is used to return the control of execution.

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

The `Expression` is returned as a value. A `return` statement must occur within a method body; otherwise a compile-time error occurs. Further, the type of the expression must match the return type of the method containing the return statement.

#### 6.1.6 Expression statements

Expression statements enumerate a set of expressions.

```
ExpressionStatement ::=  StatementExpression ';'
StatementExpression ::=  Assignment
        | MethodInvocation
        | ClassInstanceCreationExpression
```

The expressions are defined in Section 7.

#### 6.1.7 Input/Output Statements

Input and output of values of primitive types are achieved by the `input` and `output` statements.

```
IOStatement ::= (input | output) Expression ';'
```

The input statement takes an expression of primitive type, denoting a location. It reads a value of the corresponding type from the standard input and stores the value in the location. Similarly, the output statement takes an expression of primitive type, denoting a value. It writes the value of that expression onto the standard output.

The external representation of the boolean values are: `true` for true and `false` for false.

## 7 Expressions

Expressions denote rules of computation that obtain the values of variables and generate new values by applying operators.

### 7.1 Types of expressions

All expressions have a type. The type of an expression is determined by the types of its components and the semantics of its operators. The type characteristics of the operators of Java$^{--}$ are shown in table 1. Assume that `T` in the table denotes a type.

#### 7.1.1 Type conversions

Java$^{--}$ does not support explicit type conversion. However, two kinds of type conversions happen automatically. The first kind of implicit conversion applies to primitive types:

- A `char` can be used wherever an `int` is valid.

| Operator | Left Operand | Right Operand | Result |
|---|---|---|---|
| $+,-,*,/$ | int | int | int |
| Unary $-$ | int | | int |
| $<, <=, >, >=$ | int | int | boolean |
| $\&\&, \|\|$ | boolean | boolean | boolean |
| ! | boolean | | boolean |
| $==, !=$ | T | T | boolean |
| $=$ | T | T | T |

Table 1: Table of Operators

The second kind of implicit conversion involves reference conversion. A reference to an instance of a class can be used wherever a reference to an instance of a superclass of the class is used.

Examples:

```
class C { ... }
class S extends C { ...}
C c1;
S s1;

c1 = s1; // valid assignment
s1 = c1; // invalid assignment
```

Also, `null` object reference can be assigned to any reference type.

#### 7.1.2 Type equivalence

Java$^{--}$ enforces name equivalence: two types, $T_1$ and $T_2$, are equivalent if and only if they have identical names.

### 7.2 Primitive expressions

In Java$^{--}$, complex expressions are built from primitive expressions denoting variables and class members, literal constants and object references (through this or creation of new objects). A primitive expression is, thus, defined:

```
PrimitiveExpression ::=  Primary  | Name
```

## 7.3 Primary expressions

Primary expressions include most of the simplest kinds of expressions, from which all others are constructed: literals, field accesses and method invocations. A parenthesized expression is also treated syntactically as a primary expression.

```
Primary ::=  Literal
        |  this
        | '(' Expression ')'
        | ClassInstanceCreationExpression
        | FieldAccess
        | MethodInvocation
```

### 7.3.1 Literal

A literal denotes integer, string, character, boolean and null constants:

```
Literal ::=  integer_constant | string_constant | char_constant
        | null  | true | false
```

The only literal object reference is `null`. It can be used anywhere a reference is expected. `null` conventionally represents an invalid or uncreated object. Character constants appear with single quotes: 'Q'. Integer constants are strings of decimal digits. String literals appear with double quotes: "a string". `true` and `false` are boolean constants with their usual meaning.

Examples:

```
int a; char b; bool c;
Obj  o;                  // Obj is a class
a = 385;                 // 385 is integer constant
b = 'a';                 // 'a' is char constant
c = true;                // true is bool constant
o = null;                // null constant
output "String constant";// string constant
```

### 7.3.2 this

The keyword `this` either denotes a reference to the object whose method is being invoked, or to the object being constructed. It may be used only in the body of a method. Otherwise a compile-time error occurs.

An implicit `this` is added to the beginning of any field or method reference inside a method if it is not provided by the programmer. For instance, the assignment to variable `var` in

```
class Example {
    int var;
    Example1() { var = 0;}
}
```

is equivalent to the following:

```
this.var = 0;
```

`this` can be used to access an instance variable if it is hidden by a local variable or a parameter variable with the same name.

Examples:

```
class Example {
    int var;
    Example1(int var) { this.var = var; }
}
```

### 7.3.3 Class instance creation expressions

A class instance creation expression is used to create new objects that are instances of a class.

```
ClassInstanceCreationExpression ::= new ClassType  '('[ArgumentList] ')'
ArgumentList ::= Expression (',' Expression)*
```

The arguments in the argument list are used to select a constructor for the named class.

Examples:

```
Class aClass {
   int x, y
   aClass() { x = 0; y = 0;}
   aClass(int a, int b) { x = a; y = b;}
   ...
}
...
aClass var1, var2;
var1 = new aClass(0, 2);
var2 = new aClass();
...
```

Invocation of a new expression results in creation of an instance of the class on the heap. A reference for the object is returned.

### 7.3.4 Field access expressions

A field access expression is used to access a field of an object:

```
FieldAccess ::=  Primary '.' id | super '.' id
```

The type of `Primary` must be a reference type. The special form using the keyword `super` is valid only in an instance method. Also `id` must be a valid member (field or method) of the reference class denoted by `Primary` or the superclass if denoted by `super`. The following are examples of field access expressions:

Examples:

```
this.field1                 // access field1 of current object
super.fld.fld2
(new X()).i                 // field i of object (new X())
(method(a, b)).i            // field i of object returned by method(a, b)
```

### 7.3.5 Method invocation expressions

A method invocation expression is used to invoke a class or instance method:

```
MethodInvocation ::=  Name '(' [ArgumentList] ')'
        | Primary '.' id  '(' [ArgumentList] ')'
        | super '.' id '(' [ArgumentList]')'
```

The expression may contain a list of *actual parameters* which are substituted in place of their corresponding *formal parameters* defined in the method declaration. The correspondence is established by matching the positions of the parameters in actual and formal parameter lists.

Java$^{--}$ supports value parameters, where the actual parameter must be an expression (of which a variable is a simple case). The corresponding formal parameter represents a local variable of the called procedure, and the current value of the expression is initially assigned to this variable.

Examples:

```
Class ExampleClass {
  ExampleClass() {...}
  int method(ExampleClass e, int j) { ... }
  ...
}
...
int i;
ExampleClass e;
...
```

```
e = new ExampleClass();              // create an instance of Example class
i = e.method(new ExampleClass(), 4); // invoke a method on e.
```

### 7.4 Binary and unary expressions

Java$^{--}$ supports several binary and unary arithmetic and logical operators. Expression containing them are defined in the following manner:

```
Expression ::=  Expression '*' Expression | Expression '/' Expression
        | Expression '+' Expression |  Expression '-' Expression
        | Expression '&&' Expression | Expression '||' Expression
        | Expression '==' Expression | Expression '!=' Expression
        | Expression '<' Expression  | Expression '>' Expression
        | Expression '<=' Expression | Expression '>=' Expression
        | Assignment | '-' Expression | '+' Expression | '!' Expression
        | PrimitiveExpression
```

The operators have usual meaning. The precedences for the operators are described in the handout for project 2.

### 7.5 Assignment expression

Assignments are defined in the following manner:

```
Assignment ::=  LeftHandSide '=' Expression
LeftHandSide ::=  Name | FieldAccess
```

The result of the first operand of an assignment operator must be a variable, or a compile-time error occurs. This operand may be a named variable, such as a local variable or a field of the current object or class, or it may be a computed variable, as can result from a field access. The type of the assignment expression is the type of the variable.

At run time, the result of the assignment expression is the value of the variable after the assignment has occurred. A compile-time error occurs if the type of the right-hand operand cannot be converted to the type of the variable.

## 8 Notes

Much of the material in this document is taken from the following two books:

- *The Java programming language*, Ken Arnold and J. Gosling. Addison Wesley, 1995.

- *The Java language specification*, J. Gosling, K. Joy and G. Steele. Addison Wesley, 1996.