B. The *j--* Language

J-- is a subset of Java and is the language that our example compiler translates to JVM code. It has a little less than half the syntax of Java. It has classes; it has `int`s, `boolean`s, `char`s and `String`s; and it has many Java operators. The 'j' is in the name because *j--* is derived from Java; the '--' is there because *j--* has less functionality than does Java. The exercises in the text involve adding to this language. We add fragments of Java that are not already in *j--*.

B.1 A j-- *Program and its Class Declarations*

A *j--* program looks very much like a Java program. It can have an optional package statement, followed by zero or more import declarations, followed by zero or more type declarations. But in *j--*, the only kind of type declaration we have is the class declaration; *j--* has neither interfaces nor enumerations.

We may have only single-type-import declarations in *j--*; it doesn't support import-on demand declarations (e.g., `java.util.*`). The only Java types that are implicitly imported are `java.lang.Object` and `java.lang.String`. All other external Java types must be explicitly imported.

For example, the following is a legal *j--* program.

```
package pass;

import java.lang.Integer;
import java.lang.System;

public class Series
{
    public static int ARITHMETIC = 1;
    public static int GEOMETRIC = 2;
    private int a; // first term
    private int d; // common sum or multiple
    private int n; // number of terms

    public Series()
    {
        this(1, 1, 10);
    }

    public Series(int a, int d, int n)
    {
        this.a = a;
        this.d = d;
        this.n = n;
    }

    public int computeSum(int kind)
```

```
    {
        int sum = a, t = a, i = n;
        while (i-- > 1) {
            if (kind == ARITHMETIC) {
                t += d;
            }
            else if (kind == GEOMETRIC) {
                t = t * d;
            }
            sum += t;
        }
        return sum;
    }

    public static void main(String[] args)
    {
        int a = Integer.parseInt(args[0]);
        int d = Integer.parseInt(args[1]);
        int n = Integer.parseInt(args[2]);
        Series s = new Series(a, d, n);
        System.out.println("Arithmetic sum = " +
                           s.computeSum(Series.ARITHMETIC));
        System.out.println("Geometric sum = " +
                           s.computeSum(Series.GEOMETRIC));
    }
}
```

j-- is quite rich. Although *j--* is a subset of Java, it can interact with the Java API. Of course, it can only interact to the extent that it has language for talking about things in the Java API. For example, it can send messages to Java objects that take **`int`**, **`boolean`** or **`char`** arguments, and which return **`int`**, **`boolean`**, and **`char`** values, but it can not deal with **`float`**s, **`double`**s, or even **`long`**s.

As for Java, only one of the type declarations in the compilation unit (the file containing the program) can be **`public`**, and that class's **`main()`** method is the program's entry point[1]

Although *j--* does not support interface classes, it does support abstract classes. For example,

```
// Animalia.java

package pass;

import java.lang.System;

abstract class Animal
{
```

[1] A program's entry point is where the program's execution commences.

```
    protected String scientificName;

    protected Animal(String scientificName)
    {
        this.scientificName = scientificName;
    }

    public String scientificName()
    {
        return scientificName;
    }
}

class FruitFly extends Animal
{
    public FruitFly()
    {
        super("Drosophila melanogaster");
    }
}

class Tiger extends Animal
{
    public Tiger()
    {
        super("Panthera tigris corbetti");
    }
}

public class Animalia
{
    public static void main(String[] args)
    {
        FruitFly fruitFly = new FruitFly();
        Tiger tiger = new Tiger();
        System.out.println(fruitFly.scientificName());
        System.out.println(tiger.scientificName());
    }
}
```

Abstract classes in *j--* conform to the Java rules for abstract classes.

B.2. j-- Types

j-- has fewer types than does Java. This is an area where *j--* is a much smaller language than is Java.

For example, *j--* primitives include just the **int**s, **char**s, and **boolean**s. It *excludes* many of the Java primitives: **byte**, **short**, **long**, **float** and **double**.

As indicated above, *j--* has neither interfaces nor enumerations. On the other hand, *j--* has all the reference types that can be defined using classes, including the implicitly imported **`String`** and **`Object`** types.

j-- is stricter than is Java when it comes to assignment. The types of actual arguments must exactly match the types of formal parameters, and the type of right-hand side of an assignment statement must exactly match the type of the left-hand side. The only implicit conversion is the Java **`String`** conversion for the **`+`** operator; if any operand of a **`+`** is a **`String`** or if the left-hand side of a **`+=`** is a **`String`**, the other operands are converted to **`String`**s. *j--* has no other implicit conversions. But *j--* does provide casting; *j--* casting follows the same rules as Java.

That *j--* has fewer types than Java is in fact a rich source of exercises for the student. Many exercises involve the introduction of new types, and the introduction of appropriate casts, implicit type conversion and even a relaxation of the definition of *assignable*[2].

B.3. j-- *Expressions and operators*

j-- supports the following Java expressions and operators.

Expression	Operators
Assignment	`=, +=`
Conditional	`&&`
Equality	`==`
Relational	`>, <=, instanceof`[3]
Additive	`+, -`
Multiplicative	`*`
Unary (prefix)	`++, -`
Simple Unary	`!`
Postfix	`--`

It also supports casting expressions, field selection and message expressions. Both **`this`** and **`super`** may be the targets of field selection and message expressions

j-- also provides literals for the types it can talk about including **`String`**s.

[2] An object of a class or interface B is *assignable* to an object of the same class or interface, or to an object of A, which is a superclass or superinterface of B.
[3] Technically, **`instanceof`** is a keyword.

B.4. j-- *Statements and Declarations*

In addition to statement expressions[4], *j*-- provides for the **`if`** statement, **`if-else`** statement, **`while`** statement, **`return`** statement, and blocks. All of these statements follow the Java rules.

Static and instance field declarations, local variable declarations, and variable initializations are supported, and follow the Java rules.

B.5 Syntax

This section describes the lexical and syntactic grammars for the *j*-- programming language; the former specifies how individual tokens in the language are composed, and the latter specifies how language constructs are formed.

We employ the following notation to describe the grammars.

- // indicates comments;
- Non-terminals are written in the form of Java (mixed-case) identifiers;
- Terminals are written in **`bold`**;
- Token representations are enclosed in **`<>`**;
- [*x*] indicates zero or one occurrence of *x*;
- {*x*} indicates zero or more occurrences of *x*;
- *x* | *y* indicates *x* or *y*;
- ~*x* indicates negation of *x*;
- Parentheses are used for grouping;
- Level numbers in expressions indicate precedence

B.5.1 Lexical Grammar

B.5.1.1 White Space

White space in *j*-- is defined as the ASCII space (**`SP`**), horizontal tab (**`HT`**), and form feed (**`FF`**) characters, as well as line terminators: line feed (**`LF`**), carriage return (**`CR`**), and carriage return (**`CR`**) followed by line feed (**`LF`**).

B.5.1.2 Comments

j-- supports single-line comments; all the text from the ASCII characters // to the end of the line is ignored.

---

[4] A statement expression is an expression that can act as a statement. Examples include, **`i--; x = y + z;`** and **`x.doSomething();`**.

B.5.1.3 Reserved Words

The following tokens are reserved for use as keywords in *j--* and cannot be used as identifiers.

```
abstract   boolean     char       class      else       exends     false
import     instanceof  int        new        null       package    private
protected  public      return     static     super      this       true
void       while
```

B.5.1.4 Operators

The following tokens serve as operators in *j--*.

```
=      ==     >      ++     &&     <=     !      -      --     +      +=     *
```

B.5.1.5 Separators

The following tokens serve as separators in *j--*.

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

B.5.1.6 Identifiers

The following regular expression describes identifiers in *j--*.

```
<identifier> = (a-z | A-Z | _ | $) {a-z | A-Z | _ | 0-9 | $}
```

B.5.1.7 int, char and String Literals

An escape (`ESC`) character in *j--* is a `\` followed `n`, `r`, `t`, `b`, `f`, `'`, `"`, or `\`. Besides the `true`, `false`, and `null` literals, *j--* supports `int`, `char` and `String` literals as described by the following regular expressions.

```
<int_literal> = 0 | (1-9) {0-9}
<char_literal> = ' (ESC | ~(' | \ | LF | CR)) '
<string_literal> = " {ESC | ~(" | \ | LF | CR)} "
```

B.5.2 Syntactic Grammar

compilationUnit ::= [`package` qualifiedIdentifier `;`]
{`import` qualifiedIdentifier `;`}
{typeDeclaration} `EOF`

qualifiedIdentifier ::= **<identifier>** {**.** **<identifier>**}

typeDeclaration ::= modifiers classDeclaration

modifiers ::= {**public** | **protected** | **private** | **static** | **abstract**}

classDeclaration ::= **class** **<identifier>** [**extends** qualifiedIdentifier] classBody

classBody ::= **{** {modifiers memberDecl} **}**

memberDecl ::= **<identifier>** // constructor
formalParameters block
| (**void** | type) **<identiier>** // method
formalParameters (block | **;**)
| type variableDeclarators **;** // field

block ::= **{** {blockStatement} **}**

blockStatement ::= localVariableDeclarationStatement
| statement

statement ::= block
| **<identifier>** **:** statement
| **if** parExpression statement [**else** statement]
| **for** **(** forInit **;** forCond **;** forIter **)** statement
| **while** parExpression statement
| **do** statement **while** parExpression **;**
| **break** [**<identifier>**] **;**
| **continue** [**<identifier>**] **;**
| **return** [expression] **;**
| **;**
| statementExpression **;**

formalParameters ::= **(** [formalParameter {**,** formalParameter}] **)**

formalParameter ::= type **<identifier>**

parExpression ::= **(** expression **)**

localVariableDeclarationStatement ::= type variableDeclarators **;**

variableDeclarators ::= variableDeclarator {**,** variableDeclarator}

variableDeclarator ::= **<identifier>** [**=** variableInitializer]

variableInitializer ::= arrayInitializer | expression

arrayInitializer ::= `{` [variableInitializer {`,` variableInitializer}] `}`

arguments ::= `(` [expression {`,` expression}] `)`

type ::= referenceType | basicType

basicType ::= **`boolean`** | **`char`** | **`int`**

referenceType ::= basicType `[ ]` {`[ ]`}
| qualifiedIdentifier {`[ ]`}

statementExpression ::= expression // but must have side-effect, eg i++

expression ::= assignmentExpression

assignmentExpression ::= conditionalAndExpression // must be a valid lhs
[(`=` | `+=`) assignmentExpression]

conditionalAndExpression ::= equalityExpression // level 10
{`&&` equalityExpression}

equalityExpression ::= relationalExpression // level 6
{`==` relationalExpression}

relationalExpression ::= additiveExpression // level 5
[(`>` | `<=`) additiveExpression | **`instanceof`** referenceType]

additiveExpression ::= multiplicativeExpression // level 3
{(`+` | `-`) multiplicativeExpression}

multiplicativeExpression ::= unaryExpression // level 2
{`*` unaryExpression}

unaryExpression ::= `++` unaryExpression // level 1
| `-` unaryExpression
| simpleUnaryExpression

simpleUnaryExpression ::= `!` unaryExpression
| `(` basicType `)` unaryExpression //cast
| `(` referenceType `)` simpleUnaryExpression // cast
| postfixExpression

postfixExpression ::= primary {selector} {`--`}

selector ::= `.` qualifiedIdentifier [arguments]
| `[` expression `]`

primary ::= parExpression
| **`this`** [arguments]
| **`super`** (arguments | **`.`** **`<identifier>`** [arguments])
| literal
| **`new`** creator
| qualifiedIdentifier [arguments]

creator ::= (basicType | qualifiedIdentifier)
(arguments
| **`[ ]`** {**`[ ]`**} [arrayInitializer]
| newArrayDeclarator)

newArrayDeclarator ::= **`[`** expression **`]`** {**`[`** expression **`]`**} {**`[ ]`**}

literal ::= **`<int_literal>`** | **`<char_literal>`** | **`<string_literal>`** | **`true`** | **`false`** | **`null`**

B.6. The Relationship of j-- to Java

As was said earlier, *j--* is a subset of the Java programming language. Those constructs of Java that are in *j--*, roughly conform to their behavior in Java. There are several reasons for defining *j--* in this way.

- Because many students know Java, *j--* will not be totally unfamiliar.
- The exercises involve adding Java features that are not already there to *j--*. Again, because one knows Java, the behavior of these new features should be familiar.
- One learns even more about a programming language by implementing its behavior.
- Because our compiler is written in Java, the student will get more practice in Java programming.

For reasons of history and performance, most compilers are written in C or C++. One might ask, then why don't we work in one of those languages? Fair question.

Most computer science students study compilers, not because they will write compilers (although some may) but to learn how to program better: to make parts of a program work together, to learn how to apply some of the theory they have learned to their programs, and to learn how to work within an existing framework. We hope that one's working with the *j--* compiler will help in all of these areas.