

2. j-- and the Target Machine (JVM)

2.1 Introduction

The compiler that we will be discussing in this book translates a program written in a high-level programming language called *j--* into a low-level byte-coded instruction set (byte-code for short) for execution by the Java Virtual Machine (JVM).



Figure 2.1 Our Source and Our Target

In this chapter, we will provide a brief introduction to the *j--* language, the JVM, the byte-code format, and **CLEmitter** -- a high-level interface for generating the bytecode.

2.2 The *j--* programming language

j-- is a non-trivial subset of the Java programming language; it has a little less than half the syntax of Java. It has classes; it has **ints**, **booleans**, **chars** and **Strings**; 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.

2.2.1. 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't deal with **floats**, **doubles**, or even **longs**.

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 point¹

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  
{  
    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
```

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

```

{
    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.

2.2.2. *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 **ints**, **chars**, and **booleans**. 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 **Strings**. *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.

2.2.3. *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 literal for the types it can talk about including **Strings**.

2.2.4. *j--* Statements and Declarations

In addition to statement expressions⁴, *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.

2.2.5. *j--* Comments

j-- supports only Java's single-line comments.

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.

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

```
// I am a just another comment in the program.
```

Multi-line comments of the `/* ... */` variety are not supported.

2.2.6. 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.

Before going on to discuss lexical analysis, the first phase of compilation, we will first consider our ultimate target machine: the JVM. For in the end, we will want to produce code for that machine.

2.3 The JVM

The JVM is a *virtual* byte-code machine. We say it is virtual because there is no real JVM computer chip, per say, but the JVM is an abstract architecture which can have any number of implementations. For example, Sun⁵ has implemented a Java Runtime Environment (JRE) which interprets JVM programs, but uses Hotspot technology for further compiling code that is executed repeatedly to native machine code. We say it is a *byte-code* machine, because the programs it executes are sequences of bytes that represent the instructions and the operands.

⁵ Sun Microsystems, Inc. <http://www.sun.com>

Although virtual, the JVM has a definite architecture (that is, organization), and for our purposes we may consider it to be our target machine. It has an instruction set and a means for specifying operands for the instructions. And it has an internal organization.

The JVM starts up by creating an initial class, which is specified in an implementation-dependent manner, using the bootstrap class loader. The JVM then links the initial class, initializes it, and invokes its **public** class method `void main(String[] args)`. The invocation of this method drives all further execution. Execution of JVM instructions constituting the `main()` method may cause linking (and consequently creation) of additional classes and interfaces, as well as invocation of additional methods.

A JVM instruction consists of a one-byte opcode (operation code) specifying the operation to be performed, followed by zero or more operands supplying the arguments or data that are used by the operation.

The inner loop of the JVM interpreter is effectively:

```
do {
    fetch an opcode;
    if (operands) fetch operands;
    execute the action for the opcode;
} while (there is more to do);
```

The JVM defines various run-time data areas that are used during execution of a program. Some of these data areas are created on the JVM start-up and are destroyed only when the JVM exits. Other data areas are per thread⁶. Per-thread data areas are created when a thread is created and destroyed when the thread exits.

2.3.1 The pc Register

The JVM can support many threads of execution at once, and each thread has its own **pc** (program counter) register. At any point, each JVM thread is executing the code of a single method, the current method for that thread. If the method is not **native**, the **pc** register contains the address of the JVM instruction currently being executed.

2.3.2 JVM Stacks and Stack Frames

The JVM is not a register machine, but a *stack machine*. Each JVM thread has a run-time data stack, created at the same time as the thread. The JVM stack is analogous to the stack of a conventional language such as C: it holds local variables and partial results,

⁶ *j--* doesn't support implementation of multi-threaded programs.

and plays a role in method invocation and return. There are instructions for loading data values onto the stack, for performing operations on the value(s) that are on top of the stack, and there are instructions for storing the results of computations back in variables.

For example, consider the following simple albeit unlikely expression.

$$34 + 6 * 11$$

The JVM code for performing this computation follows.

```
ldc 34
ldc 6
ldc 11
imul
iadd
```

Executing this sequence of instructions takes a run-time stack through the sequence of states illustrated in Figure 2.2.

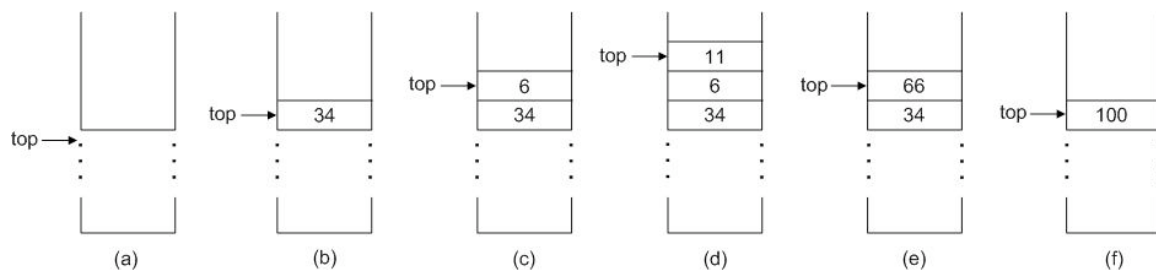


Figure 2.2 The Stack States for Computing $34 + 6 * 11$

In (a) the run-time stack is in some initial state, where top points to its top value. Executing the first `ldc` (load constant) instruction causes the JVM to push the value 34 onto the stack, leaving it in state (b). The second `ldc`, pushes 6 onto the stack, leaving it in state (c). The third `ldc` pushes 11 onto the stack, leaving it in state (d). Then, executing the `imul` (integer multiplication) instruction causes the JVM to pop the top two values (11 and 6) off from the stack, multiply them together, and to push the resultant 66 back onto the stack, leaving it in state (e). Finally, the `iadd` (integer addition) instruction pops the top two values (66 and 34) off the stack, adds them together, and pushes the resultant 100 back onto the stack, leaving it in the state (f).

As we saw in Chapter 1, the stack is organized into *stack frames*. Each time a method is invoked, a new stack frame is pushed onto the stack. All actual arguments that correspond to the method's formal parameters, and all local variables in the method's body are allocated space in this stack frame. Also, all of the method's computations, like that illustrated in Figure 2.2, are computed in this same stack frame. Computations take place in the space above the arguments and local variables.

For example, consider the following instance method, `add()`.

```
int add(int x, int y)
{
    int z;
    z = x + y;
    return z;
}
```

Now, say `add()` is a method defined in a class named `Foo`, and further assume that `f` is a variable of type `Foo`. Consider the message expression,

```
f.add(2, 3)
```

When `add()` is invoked, a stack frame like that illustrated in Figure 2.3 is pushed onto the run-time stack.

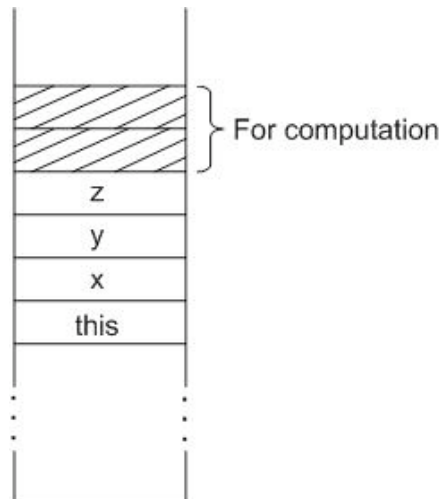


Figure 2.3 The Stack Frame for an Invocation of `add()`

Because `add()` is an instance method, the object itself, i.e. `this`, must be passed as an implicit argument in the method invocation; so `this` occupies the first location in the stack frame at offset 0. Then the actual parameter values 2 and 3, for formal parameters `x` and `y`, occupy the next two locations at offsets 1 and 2 respectively. The local variable `z` is allocated space for its value at offset 3. Finally, two locations are allocated above the parameters and local variable for the computation.

Here is a symbolic version⁷ of the code produced for `add()` by our `j--` compiler.

⁷ Produced using `javap -v Foo`.

```
int add(int, int);
```

Code:

```
Stack=2, Locals=4, Args_size=3
0:   iload_1
1:   iload_2
2:   iadd
3:   istore_3
4:   iload_3
5:   ireturn
```

Here's how the JVM executes this code.

- The `iload_1` instruction loads the integer value at offset 1 (for `x`) onto the stack at frame offset 4.
- The next `iload_2` instruction loads the integer value at offset 2 (for `y`) onto the stack at frame offset 5.
- The `iadd` instruction pops the two integers off the top of the stack (from frame offsets 4 and 5), adds them using integer addition, and then pushes the result (`x + y`) back onto the stack at frame offset 4.
- The `istore_3` instruction pops the top value (at frame offset 4) off the stack and stores it at offset 3 (for `z`).
- The `iload_3` instruction loads the value at frame offset 3 onto the stack (at frame offset 4).
- Finally, the `ireturn` instruction pops the top integer value from the stack (at frame location 4), pops the stack frame from the stack, and returns control to the invoking method, pushing the returned value onto the invoking method's frame.

Notice that the instruction set takes advantage of common offsets within the stack frame. For example, the `iload_1` instruction is really shorthand for

```
iload 1
```

The `iload_1` instruction occupies just one byte for the opcode; the opcode for `iload_1` is 27. But the other version requires two bytes: one for the opcode (21) and one byte for the operand's offset (1). The JVM is trading opcode space -- a byte may only represent up to 256 different operations -- for code space.

A frame may be extended to contain additional implementation-specific data such as the information required to support a run-time debugger.

2.3.3. The Heap

Objects are represented on the stack as pointers into the *heap*, which is shared among all JVM threads. The heap is the run-time data area from which memory for all class

instances and arrays is allocated. It is created during the JVM start-up. Heap storage for objects is reclaimed by an automatic storage management system called the *garbage collector*.

2.3.4. The Method Area

The JVM has a method area that is shared among all JVM threads. It is created during the JVM start-up. It stores per-class structures such as the run-time constant pool, field and method data, and the code for methods and constructors, including the special methods used in class and instance initialization and interface type initialization.

2.3.5. The Runtime Constant Pool

The *run-time constant pool* is a per-class or per-interface run-time representation of the `constant_pool` table in a class file. It contains several kinds of constants, ranging from numeric literals known at compile time to method and field references that must be resolved at runtime. It is constructed when the class or interface is created by the JVM.

2.3.6. Abrupt Method Invocation Completion

A method invocation completes abruptly if execution of a JVM instruction within the method causes the JVM to throw an exception, and that exception is not handled within the method. Execution of an `ATHROW` instruction also causes an exception to be explicitly thrown and, if the exception is not caught by the current method, results in abrupt method invocation completion. A method invocation that completes abruptly never returns a value to its invoker.

2.4 The Class File

The byte-code that *j*-compiler generates from a source program is stored in a binary file with a `.class` extension. We refer to such files as *class files*. Each class file contains the definition of a single class or interface⁸. A class file consists of a stream of 8-bit bytes. All 16-bit, 32-bit, and 64-bit quantities are constructed by reading in two, four, and eight consecutive 8-bit bytes, respectively. Multi-byte data items are always stored in big-endian order, where the high bytes come first.

⁸ *j*-- currently does not support interfaces.

A class file consists of a single **ClassFile** structure⁹:

```

ClassFile {
    u4 magic;
    u2 minor_version;
    u2 major_version;
    u2 constant_pool_count;
    cp_info constant_pool[constant_pool_count-1];
    u2 access_flags;
    u2 this_class;
    u2 super_class;
    u2 interfaces_count;
    u2 interfaces[interfaces_count];
    u2 fields_count;
    field_info fields[fields_count];
    u2 methods_count;
    method_info methods[methods_count];
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}

```

The types **u1**, **u2**, and **u4** represent an unsigned one-, two-, or four-byte quantity, respectively. The items of the **ClassFile** structure are described below.

Magic	A magic number (0xCAFEBABE) identifying the class file format.
minor_version, major_version	Together, a major and minor version number determine the version of the class file format. A JVM implementation can support a class file format of version <i>v</i> if and only <i>v</i> lies in some contiguous range of versions. Only Sun specifies the range of versions a JVM implementation may support.
constant_pool_count	Number of entries in the constant_pool table plus one.
constant_pool[]	A table of structures representing various string constants, class and interface names, field names, and other constants that are referred to within the ClassFile structure and its substructures.
access_flags	Mask of flags used to denote access permissions to and properties of this class

⁹ We use C-like syntax to describe the structure. The data types used within are just representational, and do not correspond to types in C, Java, or *j--*.

	or interface.
this_class	Must be a valid index into the constant_pool table. The entry at that index must be a structure representing the class or interface defined by this class file.
super_class	Must be a valid index into the constant_pool table. The entry at that index must be the structure representing the direct superclass of the class or interface defined by this class file.
interfaces_count	The number of direct super interfaces of the class or interface defined by this class file.
interfaces[]	Each value in the table must be a valid index into the constant_pool table. The entry at each index must be a structure representing an interface that is a direct superinterface of the class or interface defined by this class file.
fields_count	Number of entries in the fields table.
fields[]	Each value in the table must be a field_info structure giving complete description of a field in the class or interface defined by this class file.
methods_count	Number of entries in the methods table.
methods[]	Each value in the table must be a method_info structure giving complete description of a method in the class or interface defined by this class file.
attributes_count	Number of entries in the attributes table.
attributes[]	Must be a table of class attributes.

The internals for all of these are fully described in (Lindholm and Yellin, 1999).

One may certainly create class files by directly working with a binary output stream. However, this approach is rather arcane, and involves a tremendous amount of housekeeping; one has to maintain a representation for the **constant_pool** table, the program counter **pc**, compute branch offsets, compute stack depths, perform various bitwise operations, and do much more.

It would be much easier if there were a high level interface that would abstract out the gory details of the class file structure. The **CLEmitter** does exactly this.

2.5 The **CLEmitter**

The *j--* compiler's purpose is to produce a class file. Given the complexity of class files we supply a tool called the **CLEmitter** to ease the generation of code and the creation of class files.

The **CLEmitter**¹⁰ has a relatively small set of methods that support

- the creation of a class or an interface;
- the addition of fields and methods to the class;
- the addition of instructions, exception handlers, and code attributes to methods;
- the addition of inner classes;
- optional field, method, and class attributes;
- checking for errors; and
- the writing of the class file to the file system.

While it is much simpler to work with an interface like **CLEmitter**, one still has to be aware of certain aspects of the target machine, such as the instruction set.

Figure 2.4 outlines the necessary steps for creating an in-memory representation of a class file using the **CLEmitter** interface, and then writing that class file to the file system.

¹⁰ This is a class in the **jminusminus** package under `$j/j--/src` folder. The classes that **CLEmitter** depends on are also in that package and have a **CL** prefix.

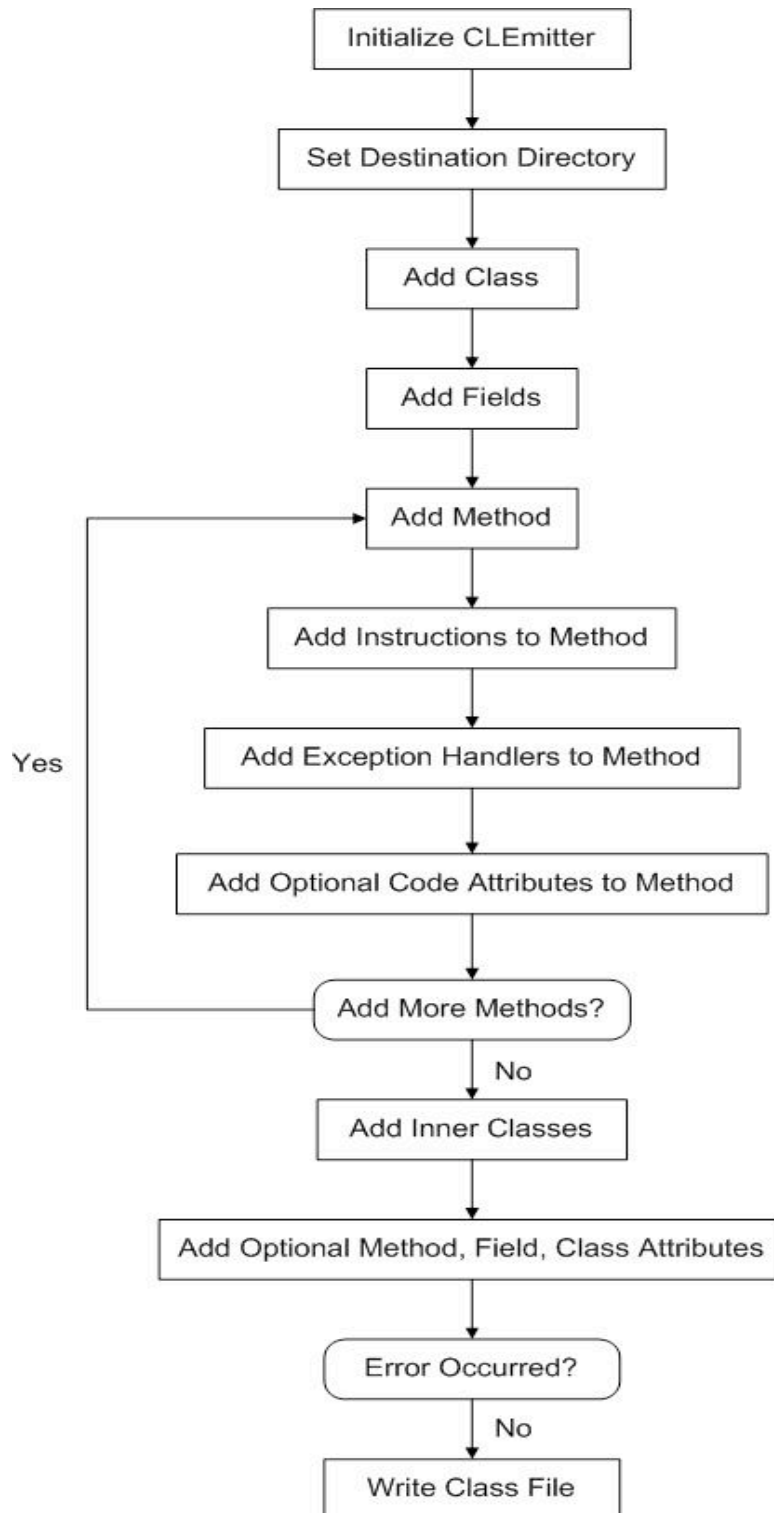


Figure 2.4: A Recipe for Creating a Class File

2.5.1 The CLEmitter Interface

To instantiate a `CLEmitter`, one simply invokes its constructor.

```
CLEmitter output = new CLEmitter();
```

To set the destination directory, one invokes `destinationDir()`.

```
output.destinationDir("/home/swamir/classes");
```

One then goes about adding classes, method, fields and instructions. There are methods for adding all of these.

For example, one may add a `class` to the class file using something like,

```
accessFlags.add("public");
superInterfaces = null;
isSynthetic = true;
output.addClass(accessFlags, "Factorial", "java/lang/Object",
                superInterfaces, isSynthetic);
```

where `accessFlags`¹¹ is a list of class access and property flags, `"Factorial"` is the name of the class in internal form¹², `"java/lang/Object"` is the name of the super class in internal form, `superInterfaces` is a list of direct super interfaces of the class in internal form, and `isSynthetic` specifies whether the class appears in source code or was synthesized by the compiler.

One may add a method using,

```
accessFlags.add("public");
output.addMethod(accessFlags, "factorial", "(I)I", exceptions,
                 isSynthetic);
```

where `accessFlags` is a list of method access and property flags, `"factorial"` is the name¹³ of the method, `"(I)I"` is the method descriptor, `exceptions` is a list of

¹¹ Note that the `CLEmitter` expects the access and property flags as `Strings` and internally translates them to a mask of flags. For example, "public" is translated to `ACC_PUBLIC (0x0001)`.

¹² Fully qualified form, with identifiers separated by `"/"`.

¹³ Instance constructors must have the name `<init>` and static constructors must have the name `<clinit>`.

exceptions in internal form that this method throws, and `isSynthetic` specifies whether the method appears in source code or was synthesized by the compiler.

A comment on the method descriptor is warranted. The *method descriptor* describes the method's signature in a format internal to the JVM. The `I` is the internal type descriptor for the primitive type `int`, so the

```
(I)I
```

specifies that the method takes one integer argument, and returns a value having integer type.

`CLEmitter` supports emitting all kinds of JVM instructions. For example, a no-argument instruction (one that does not expect any operands on the stack) may be generated with,

```
output.addNoArgInstruction(ILOAD_1);
```

where `ILOAD_1` is the opcode.

A more detailed discussion of descriptors and the `CLEmitter` interface may be found in Appendix C.

2.5.2 CLEmitter Examples

The `j--` compiler uses the `CLEmitter` to construct class files from abstract syntax trees that are built from `j--` programs. The invocation of `CLEmitter`'s methods are distributed across the `codegen()` methods defined for the various node objects in the AST. But `CLEmitter` can stand on its own, as the following two examples illustrate.

2.5.2.1 IntFactorial -- a `j--` program

In our first example, we deal with the following `j--` program.

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

public class IntFactorial
{
    public int factorial(int n)
    {
        if (n <= 1) {
            return n;
        }
        else {
```

```

        return (n * factorial(n - 1));
    }
}

public static void main(String[] args)
{
    IntFactorial f = new IntFactorial();
    int n = Integer.parseInt(args[ 0 ]);
    System.out.println("(int) factorial(" + n + ") = " +
        f.factorial(n));
}
}

```

If we were to compile this program with *j--*, and then disassemble the class file using,

```

j-- IntFactorial.java
javap -v IntFactorial

```

we would get the following.

```

public class IntFactorial extends java.lang.Object
  minor version: 0
  major version: 49
  Constant pool:
const #1 = Asciz      IntFactorial;
const #2 = class     #1;      // IntFactorial
const #3 = Asciz      java/lang/Object;
const #4 = class     #3;      // java/lang/Object
const #5 = Asciz      <init>;
const #6 = Asciz      ()V;
const #7 = NameAndType #5:#6;// "<init>":()V
const #8 = Method     #4.#7;  // java/lang/Object."<init>":()V
const #9 = Asciz      Code;
const #10 = Asciz     factorial;
const #11 = Asciz     (I)I;
const #12 = NameAndType #10:#11;// factorial:(I)I
const #13 = Method     #2.#12; // IntFactorial.factorial:(I)I
const #14 = Asciz     main;
const #15 = Asciz     ([Ljava/lang/String;)V;
const #16 = Method     #2.#7;  // IntFactorial."<init>":()V
const #17 = Asciz     java/lang/Integer;
const #18 = class     #17;    // java/lang/Integer
const #19 = Asciz     parseInt;
const #20 = Asciz     (Ljava/lang/String;)I;
const #21 = NameAndType #19:#20;//
parseInt:(Ljava/lang/String;)I
const #22 = Method     #18.#21;      //
java/lang/Integer.parseInt:(Ljava/lang/String;)I
const #23 = Asciz     java/lang/System;

```

2-18

```

const #24 = class      #23;    // java/lang/System
const #25 = Asciz     out;
const #26 = Asciz     Ljava/io/PrintStream;;
const #27 = NameAndType #25:#26;// out:Ljava/io/PrintStream;
const #28 = Field     #24.#27;    //
java/lang/System.out:Ljava/io/PrintStream;
const #29 = Asciz     java/lang/StringBuilder;
const #30 = class    #29;    // java/lang/StringBuilder
const #31 = Method    #30.#7; //
java/lang/StringBuilder."<init>":()V
const #32 = Asciz     (int) factorial(;
const #33 = String    #32;    // (int) factorial(
const #34 = Asciz     append;
const #35 = Asciz     (Ljava/lang/String;)Ljava/lang/StringBuilder;;
const #36 = NameAndType #34:#35;//
append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
const #37 = Method    #30.#36;    //
java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/Str
ingBuilder;
const #38 = Asciz     (I)Ljava/lang/StringBuilder;;
const #39 = NameAndType #34:#38;//
append:(I)Ljava/lang/StringBuilder;
const #40 = Method    #30.#39;    //
java/lang/StringBuilder.append:(I)Ljava/lang/StringBuilder;
const #41 = Asciz     ) = ;
const #42 = String    #41;    // ) =
const #43 = Asciz     toString;
const #44 = Asciz     ()Ljava/lang/String;;
const #45 = NameAndType #43:#44;// toString:()Ljava/lang/String;
const #46 = Method    #30.#45;    //
java/lang/StringBuilder.toString:()Ljava/lang/String;
const #47 = Asciz     java/io/PrintStream;
const #48 = class    #47;    // java/io/PrintStream
const #49 = Asciz     println;
const #50 = Asciz     (Ljava/lang/String;)V;
const #51 = NameAndType #49:#50;// println:(Ljava/lang/String;)V
const #52 = Method    #48.#51;    //
java/io/PrintStream.println:(Ljava/lang/String;)V

{
public IntFactorial();
Code:
Stack=1, Locals=1, Args_size=1
0:   aload_0
1:   invokespecial #8; //Method
java/lang/Object."<init>":()V
4:   return

public int factorial(int);

```

Code:

```
Stack=4, Locals=2, Args_size=2
0:   iload_1
1:   iconst_1
2:   if_icmpgt      10
5:   iload_1
6:   ireturn
7:   goto      20
10:  iload_1
11:  aload_0
12:  iload_1
13:  iconst_1
14:  isub
15:  invokevirtual   #13; //Method factorial:(I)I
18:  imul
19:  ireturn
20:  nop
```

```
public static void main(java.lang.String[]);
```

Code:

```
Stack=4, Locals=3, Args_size=1
0:   new      #2; //class IntFactorial
3:   dup
4:   invokespecial #16; //Method "<init>":()V
7:   astore_1
8:   aload_0
9:   iconst_0
10:  aaload
11:  invokestatic  #22; //Method
java/lang/Integer.parseInt:(Ljava/lang/String;)I
14:  istore_2
15:  getstatic    #28; //Field
java/lang/System.out:Ljava/io/PrintStream;
18:  new      #30; //class java/lang/StringBuilder
21:  dup
22:  invokespecial #31; //Method
java/lang/StringBuilder."<init>":()V
25:  ldc      #33; //String (int) factorial(
27:  invokevirtual #37; //Method
java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/Str
ingBuilder;
30:  iload_2
31:  invokevirtual #40; //Method
java/lang/StringBuilder.append:(I)Ljava/lang/StringBuilder;
34:  ldc      #42; //String ) =
36:  invokevirtual #37; //Method
java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/Str
ingBuilder;
39:  aload_1
40:  iload_2
```

2-20

```

    41:  invokevirtual   #13; //Method factorial:(I)I
    44:  invokevirtual   #40; //Method
java/lang/StringBuilder.append:(I)Ljava/lang/StringBuilder;
    47:  invokevirtual   #46; //Method
java/lang/StringBuilder.toString:()Ljava/lang/String;
    50:  invokevirtual   #52; //Method
java/io/PrintStream.println:(Ljava/lang/String;)V
    53:  return
}

```

The following (Java) program makes direct use of **CLEmitter** for generating this same code. It emits exactly the same code that the *j--* compiler emits for **IntFactorial** using exactly the same **CLEmitter** methods invoked by the *j--* compiler.

```

import jminusminus.CLEmitter;
import static jminusminus.CLConstants.*;
import java.util.ArrayList;

public class GenIntFactorial
{
    public static void main(String[] args)
    {
        CLEmitter e = new CLEmitter();
        ArrayList<String> accessFlags =
            new ArrayList<String>();

        // Add IntFactorial class
        accessFlags.add("public");
        e.addClass(accessFlags, "IntFactorial",
            "java/lang/Object", null, true);

        // Add the implicit no-arg constructor IntFactorial()
        accessFlags.clear();
        accessFlags.add("public");
        e.addMethod(accessFlags, "<init>", "()V", null, true);
        e.addNoArgInstruction(ALOAD_0);
        e.addMemberAccessInstruction(INVOKESPECIAL,
            "java/lang/Object", "<init>", "()V");
        e.addNoArgInstruction(RETURN);

        // Add factorial() method to IntFactorial
        accessFlags.clear();
        accessFlags.add("public");
        e.addMethod(accessFlags, "factorial", "(I)I", null,
            true);
        e.addNoArgInstruction(ILOAD_1);
        e.addNoArgInstruction(ICONST_1);
        e.addBranchInstruction(IF_ICMPGT, "falseLabel");
    }
}

```

```

e.addNoArgInstruction(ILOAD_1);
e.addNoArgInstruction(IRETURN);
e.addLabel("falseLabel");
e.addNoArgInstruction(ILOAD_1);
e.addNoArgInstruction(ALOAD_0);
e.addNoArgInstruction(ILOAD_1);
e.addNoArgInstruction(ICONST_1);
e.addNoArgInstruction(ISUB);
e.addMemberAccessInstruction(INVOKEVIRTUAL,
    "IntFactorial",
    "factorial", "(I)I");
e.addNoArgInstruction(IMUL);
e.addNoArgInstruction(IRETURN);

// Add main() method to IntFactorial
accessFlags.clear();
accessFlags.add("public");
accessFlags.add("static");
e.addMethod(accessFlags, "main",
    "([Ljava/lang/String;)V", null, true);
e.addReferenceInstruction(NEW, "IntFactorial");
e.addNoArgInstruction(DUP);
e.addMemberAccessInstruction(INVOKESPECIAL,
    "IntFactorial", "<init>", "()V");
e.addNoArgInstruction(ASTORE_1);
e.addNoArgInstruction(ALOAD_0);
e.addNoArgInstruction(ICONST_0);
e.addNoArgInstruction(AALOAD);
e.addMemberAccessInstruction(INVOKESTATIC,
    "java/lang/Integer", "parseInt",
    "(Ljava/lang/String;)I");
e.addNoArgInstruction(ISTORE_2);
e.addMemberAccessInstruction(GETSTATIC,
    "java/lang/System", "out", "Ljava/io/PrintStream;");
e.addReferenceInstruction(NEW,
    "java/lang/StringBuffer");
e.addNoArgInstruction(DUP);
e.addMemberAccessInstruction(INVOKESPECIAL,
    "java/lang/StringBuffer", "<init>", "()V");
e.addLDCInstruction("(int) factorial()");
e.addMemberAccessInstruction(INVOKEVIRTUAL,
    "java/lang/StringBuffer", "append",
    "(Ljava/lang/String;)Ljava/lang/StringBuffer;");
e.addNoArgInstruction(ILOAD_2);
e.addMemberAccessInstruction(INVOKEVIRTUAL,
    "java/lang/StringBuffer",
    "append", "(I)Ljava/lang/StringBuffer;");
e.addLDCInstruction(") = ");
e.addMemberAccessInstruction(INVOKEVIRTUAL,
    "java/lang/StringBuffer", "append",

```

```

        "(Ljava/lang/String;)Ljava/lang/StringBuffer;");
e.addNoArgInstruction(ALOAD_1);
e.addNoArgInstruction(ILOAD_2);
e.addMemberAccessInstruction(INVOKEVIRTUAL,
    "IntFactorial", "factorial", "(I)I");
e.addMemberAccessInstruction(INVOKEVIRTUAL,
    "java/lang/StringBuffer",
    "append", "(I)Ljava/lang/StringBuffer;");
e.addMemberAccessInstruction(INVOKEVIRTUAL,
    "java/lang/StringBuffer",
    "toString", "()Ljava/lang/String;");
e.addMemberAccessInstruction(INVOKEVIRTUAL,
    "java/io/PrintStream",
    "println", "(Ljava/lang/String;)V");
e.addNoArgInstruction(RETURN);

// Write IntFactorial.class to file system
e.write();
}
}

```

2.5.2.1 LongFactorial -- a Java program

Consider the following (Java) program, which computes factorial for (two-word) **long** values. Notice that this cannot be a *j--* program because it uses type **long**, which is not in the *j--* language.

```

public class LongFactorial
{
    public long factorial(long n)
    {
        if (n <= 1) {
            return n;
        }
        else {
            return (n * factorial(n - 1));
        }
    }

    public static void main(String[] args)
    {
        try {
            LongFactorial f = new LongFactorial();
            long n = Long.parseLong(args[ 0 ]);
            System.out.println("Factorial(" + n + ") = " +
                f.factorial(n));
        }
        catch (NumberFormatException e) {

```

2-23

```

        System.err.println("Invalid number " + args[ 0 ]);
    }
}
}

```

Say we wanted to add the `long` type to *j--*. To see what kind of code we need to generate for `longs`, we could compile this program using the Sun *javac* compiler and then use *javap* to get a readable version,

```

javac LongFactorial.java
javap -v LongFactorial

```

That would produce the following JVM code. We have removed the constant pool from the listing to make it more succinct.

```

{
public LongFactorial();
  Code:
    Stack=1, Locals=1, Args_size=1
    0:  aload_0
    1:  invokespecial  #1; //Method
java/lang/Object.<init>:()V
    4:  return
 LineNumberTable:
    line 1: 0

public long factorial(long);
  Code:
    Stack=7, Locals=3, Args_size=2
    0:  lload_1
    1:  lconst_1
    2:  lcmp
    3:  ifgt      8
    6:  lload_1
    7:  lreturn
    8:  lload_1
    9:  aload_0
   10:  lload_1
   11:  lconst_1
   12:  lsub
   13:  invokevirtual  #2; //Method factorial:(J)J
   16:  lmul
   17:  lreturn
 LineNumberTable:
    line 5: 0
    line 6: 6
    line 9: 8

public static void main(java.lang.String[]);

```

2-24

```

Code:
  Stack=5, Locals=4, Args_size=1
  0:   new      #3; //class LongFactorial
  3:   dup
  4:   invokespecial  #4; //Method "<init>":()V
  7:   astore_1
  8:   aload_0
  9:   iconst_0
 10:  aaload
 11:  invokestatic   #5; //Method
java/lang/Long.parseLong:(Ljava/lang/String;)J
 14:  lstore_2
 15:  getstatic      #6; //Field
java/lang/System.out:Ljava/io/PrintStream;
 18:  new      #7; //class java/lang/StringBuilder
 21:  dup
 22:  invokespecial  #8; //Method
java/lang/StringBuilder."<init>":()V
 25:  ldc      #9; //String Factorial(
 27:  invokevirtual #10; //Method
java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/Str
ingBuilder;
 30:  lload_2
 31:  invokevirtual #11; //Method
java/lang/StringBuilder.append:(J)Ljava/lang/StringBuilder;
 34:  ldc      #12; //String ) =
 36:  invokevirtual #10; //Method
java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/Str
ingBuilder;
 39:  aload_1
 40:  lload_2
 41:  invokevirtual #2; //Method factorial:(J)J
 44:  invokevirtual #11; //Method
java/lang/StringBuilder.append:(J)Ljava/lang/StringBuilder;
 47:  invokevirtual #13; //Method
java/lang/StringBuilder.toString:()Ljava/lang/String;
 50:  invokevirtual #14; //Method
java/io/PrintStream.println:(Ljava/lang/String;)V
 53:  goto     84
 56:  astore_1
 57:  getstatic   #16; //Field
java/lang/System.err:Ljava/io/PrintStream;
 60:  new      #7; //class java/lang/StringBuilder
 63:  dup
 64:  invokespecial  #8; //Method
java/lang/StringBuilder."<init>":()V
 67:  ldc      #17; //String Invalid number
 69:  invokevirtual #10; //Method
java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/Str
ingBuilder;

```

2-25

```

72:  aload_0
73:  iconst_0
74:  aaload
75:  invokevirtual #10; //Method
java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/Str
ingBuilder;
78:  invokevirtual #13; //Method
java/lang/StringBuilder.toString:()Ljava/lang/String;
81:  invokevirtual #14; //Method
java/io/PrintStream.println:(Ljava/lang/String;)V
84:  return
Exception table:
  from   to  target type
    0     53    56    Class java/lang/NumberFormatException

LineNumberTable:
  line 16: 0
  line 17: 8
  line 18: 15
  line 23: 53
  line 21: 56
  line 22: 57
  line 24: 84
}

```

The following (Java) program makes direct use of **CLEmitter** for generating this same code. It emits exactly the same¹⁴ code that the *javac* compiler emits for **LongFactorial** using exactly the same **CLEmitter** methods invoked by the *javac* compiler.

```

import jminusminus.CLEmitter;
import static jminusminus.CLConstants.*;
import java.util.ArrayList;

public class GenLongFactorial
{
    public static void main(String[] args)
    {
        CLEmitter e = new CLEmitter();
        ArrayList<String> accessFlags = new ArrayList<String>();

        // Add LongFactorial class
        accessFlags.add("public");
        e.addClass(accessFlags, "LongFactorial",
            "java/lang/Object", null, true);
    }
}

```

¹⁴ Not *exactly* the same. The Sun *javac* compiler produces line number information that **GenLongFactorial** does not.

```

// Add the implicit no-arg constructor LongFactorial()
accessFlags.clear();
accessFlags.add("public");
e.addMethod(accessFlags, "<init>", "()V", null, true);
e.addNoArgInstruction(ALOAD_0);
e.addMemberAccessInstruction(INVOKEESPECIAL,
    "java/lang/Object", "<init>", "()V");
e.addNoArgInstruction(RETURN);

// Add factorial() method to LongFactorial
accessFlags.clear();
accessFlags.add("public");
e.addMethod(accessFlags, "factorial", "(J)J", null,
    true);
e.addNoArgInstruction(LLOAD_1);
e.addNoArgInstruction(LCONST_1);
e.addNoArgInstruction(LCMP);
e.addBranchInstruction(IFGT, "falseLabel");
e.addNoArgInstruction(LLOAD_1);
e.addNoArgInstruction(LRETURN);
e.addLabel("falseLabel");
e.addNoArgInstruction(LLOAD_1);
e.addNoArgInstruction(ALOAD_0);
e.addNoArgInstruction(LLOAD_1);
e.addNoArgInstruction(LCONST_1);
e.addNoArgInstruction(LSUB);
e.addMemberAccessInstruction(INVOKEVIRTUAL,
    "LongFactorial", "factorial", "(J)J");
e.addNoArgInstruction(LMUL);
e.addNoArgInstruction(LRETURN);

// Add main() method to LongFactorial
accessFlags.clear();
accessFlags.add("public");
accessFlags.add("static");
e.addMethod(accessFlags, "main",
    "([Ljava/lang/String;)V", null, true);
e.addExceptionHandler("tryStart", "tryEnd", "catch",
    "java/lang/NumberFormatException");
e.addLabel("tryStart");
e.addReferenceInstruction(NEW, "LongFactorial");
e.addNoArgInstruction(DUP);
e.addMemberAccessInstruction(INVOKEESPECIAL,
    "LongFactorial", "<init>", "()V");
e.addNoArgInstruction(ASTORE_1);
e.addNoArgInstruction(ALOAD_0);
e.addNoArgInstruction(ICONST_0);
e.addNoArgInstruction(AALOAD);
e.addMemberAccessInstruction(INVOKESTATIC,

```

```

        "java/lang/Long", "parseLong",
        "(Ljava/lang/String;)J");
e.addNoArgInstruction(LSTORE_2);
e.addMemberAccessInstruction(GETSTATIC,
    "java/lang/System",
    "out", "Ljava/io/PrintStream;");
e.addReferenceInstruction(NEW,
    "java/lang/StringBuffer");
e.addNoArgInstruction(DUP);
e.addMemberAccessInstruction(INVOKESPECIAL,
    "java/lang/StringBuffer", "<init>", "()V");
e.addLDCInstruction("Factorial(");
e.addMemberAccessInstruction(INVOKEVIRTUAL,
    "java/lang/StringBuffer", "append",
    "(Ljava/lang/String;)Ljava/lang/StringBuffer;");
e.addNoArgInstruction(LLOAD_2);
e.addMemberAccessInstruction(INVOKEVIRTUAL,
    "java/lang/StringBuffer", "append",
    "(J)Ljava/lang/StringBuffer;");
e.addLDCInstruction(") = ");
e.addMemberAccessInstruction(INVOKEVIRTUAL,
    "java/lang/StringBuffer", "append",
    "(Ljava/lang/String;)Ljava/lang/StringBuffer;");
e.addNoArgInstruction(ALOAD_1);
e.addNoArgInstruction(LLOAD_2);
e.addMemberAccessInstruction(INVOKEVIRTUAL,
    "LongFactorial", "factorial", "(J)J");
e.addMemberAccessInstruction(INVOKEVIRTUAL,
    "java/lang/StringBuffer", "append",
    "(J)Ljava/lang/StringBuffer;");
e.addMemberAccessInstruction(INVOKEVIRTUAL,
    "java/lang/StringBuffer", "toString",
    "()Ljava/lang/String;");
e.addMemberAccessInstruction(INVOKEVIRTUAL,
    "java/io/PrintStream", "println",
    "(Ljava/lang/String;)V");
e.addLabel("tryEnd");
e.addBranchInstruction(GOTO, "done");
e.addLabel("catch");
e.addNoArgInstruction(ASTORE_1);
e.addMemberAccessInstruction(GETSTATIC,
    "java/lang/System",
    "err", "Ljava/io/PrintStream;");
e.addReferenceInstruction(NEW,
    "java/lang/StringBuffer");
e.addNoArgInstruction(DUP);
e.addMemberAccessInstruction(INVOKESPECIAL,
    "java/lang/StringBuffer", "<init>", "()V");
e.addLDCInstruction("Invalid number ");
e.addMemberAccessInstruction(INVOKEVIRTUAL,

```

```

        "java/lang/StringBuffer", "append",
        "(Ljava/lang/String;)Ljava/lang/StringBuffer;");
e.addNoArgInstruction(ALOAD_0);
e.addNoArgInstruction(ICONST_0);
e.addNoArgInstruction(AALOAD);
e.addMemberAccessInstruction(INVOKEVIRTUAL,
    "java/lang/StringBuffer", "append",
    "(Ljava/lang/String;)Ljava/lang/StringBuffer;");
e.addMemberAccessInstruction(INVOKEVIRTUAL,
    "java/lang/StringBuffer", "toString",
    "()Ljava/lang/String;");
e.addMemberAccessInstruction(INVOKEVIRTUAL,
    "java/io/PrintStream", "println",
    "(Ljava/lang/String;)V");
e.addLabel("done");
e.addNoArgInstruction(RETURN);

// Write LongFactorial.class to file system
e.write();
}
}

```

If we were to add `longs` to `j--`, we would have to modify our `j--` compiler to invoke these same methods on `CLEmitter`.

Further Reading

Much more information about both the JVM and class files may be found in the JVM Specification (Lindholm and Yellin, 1999). See Chapter 3 of the specification for detailed information about the JVM's structure. See Chapter 4 for a detailed description of the class file format. See Chapter 7 for hints on compiling various Java language constructs to JVM bytecode.

Exercises

- 2.1 Write a `j--` program `Fibonacci.java` that accepts a number n as input, and outputs the n th Fibonacci number.
- 2.2 Write a `j--` program `GCD.java` that accepts two numbers a and b as input, and outputs the Greatest Common Divisor (GCD) of a and b . Hint: Use the Euclidean algorithm¹⁵.

¹⁵ http://en.wikipedia.org/wiki/Euclidean_algorithm

- 2.3 Write a *j--* program **Primes.java** that accepts a number *n* as input, and outputs the all the prime numbers that are less than or equal to *n*. Hint: Use the Sieve of Eratosthenes algorithm¹⁶. For example:

```
java Primes 11
```

should output:

```
2 3 5 7 11
```

- 2.4 Write a *j--* program **Date.java** that accepts a date in “yyyy-mm-dd” format as input, and outputs the date in “Month Day, Year” format. For example:

```
java Date 1879-03-1417
```

should output:

```
March 14, 1879
```

- 2.5 Write a *j--* program **Palindrome.java** that accepts a string as input, and outputs the string if it is a palindrome (a string that reads the same in either direction), and outputs nothing if it isn't. The program should be case-insensitive to the input. For example:

```
java Palindrome Malayalam18
```

should output:

```
Malayalam
```

- 2.6 Suggest enhancements to the *j--* language that would simplify the implementation of the programs described in the previous exercises.
- 2.7 For each of the *j--* programs described in exercises 2.1 to 2.5, write a JUnit test case and integrate it with the *j--* test framework (Appendix C describes how this can be done).
- 2.8 Disassemble (Appendix A describes how this can be done) a Java class (say **java.util.ArrayList**), study the output, and list the following:

- Major and minor version

¹⁶ http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes

¹⁷ Day Albert Einstein was born

¹⁸ Language spoken in Kerala, a southern Indian state

- Size of the constant pool table
- Super class
- Interfaces
- Field names, their access modifiers, type descriptors, and their attributes (just names)
- Method names, their access modifiers, descriptors, exceptions thrown, and their method and code attributes (just names)
- Class attributes (just names)

2.9 Compile `$j/j--/tests/pass/HelloWorld.java`¹⁹ using the *j--* compiler and Sun's *javac* compiler. Disassemble the class file produced by each and compare the output. What differences do you see?

2.10 Disassemble the class file produced by the *j--* compiler for `$j/j--/tests/pass/Series.java`, save the output in `Series.bytecode`. Add a single-line (`// ...`) comment for each JVM instruction in `Series.bytecode` explaining what the instruction does.

2.11 Write down the following class names in internal form:

```
java.lang.Thread
java.util.ArrayList
java.io.FileNotFoundException
jminusminus.Parser
Employee
```

2.12 Write down the method descriptor for each of the following constructors/method declarations:

```
public Employee(String name) ...
public Coordinates(float latitude, float longitude) ...
public Object get(String key) ...
public void put(String key, Object o) ...
public static int[] sort(int[] n, boolean ascending) ...
public int[][] transpose(int[][] matrix) ...
```

2.13 Write a program `GenGCD.java` that produces, using `CLEmitter`, a `GCD.class` file with the following methods:

```
// Returns the Greatest Common Divisor (GCD) of a and b.
public static int compute(int a, int b)
{
    ...
}
```

¹⁹ `$j` is the directory containing the *j--* source tree.

Running GCD as follows:

```
java GCD 42 84
```

should output:

```
42
```

Modify `GenGCD.java` to handle `java.lang.NumberFormatException` that `Integer.parseInt()` raises if the argument is not an integer, and in the handler, print an appropriate error message to `STDERR`.

- 2.14 Write a program `GenPrimality.java` that produces, using `CLEmitter`, `Primality.class`, `Primality1.class`, `Primality2.class`, and `Primality3.class` files, where `Primality.class` is an interface with the following method:

```
// Returns true if the specified number is prime, false
// otherwise.
public boolean isPrime(int n);
```

and `Primality1.class`, `Primality2.class`, and `Primality3.class` are three different implementations of the interface. Write a `j--` program `TestPrimality.java` that has a test driver for the three implementations.

- 2.15 Write a program `GenWC.java` that produces, using `CLEmitter`, a `WC.class` which emulates the UNIX command `wc` that displays the number of lines, words, and bytes contained in each input file.
- 2.16 Write a program `GenGravity.java` that produces, using `CLEmitter`, a `Gravity.class` file which computes the acceleration g due to gravity at a point on the surface of a massive body. The program should accept the mass M of the body, and the distance r of the point from body's center as input. Use the following formula for computing g :

$$g = GMr^{-2}$$

where $G = 6.67 \times 10^{-11} \text{ Nm}^2\text{kg}^{-2}$, is the universal gravitational constant.

