CS 326
Programming Languages

The Espresso Virtual Machine

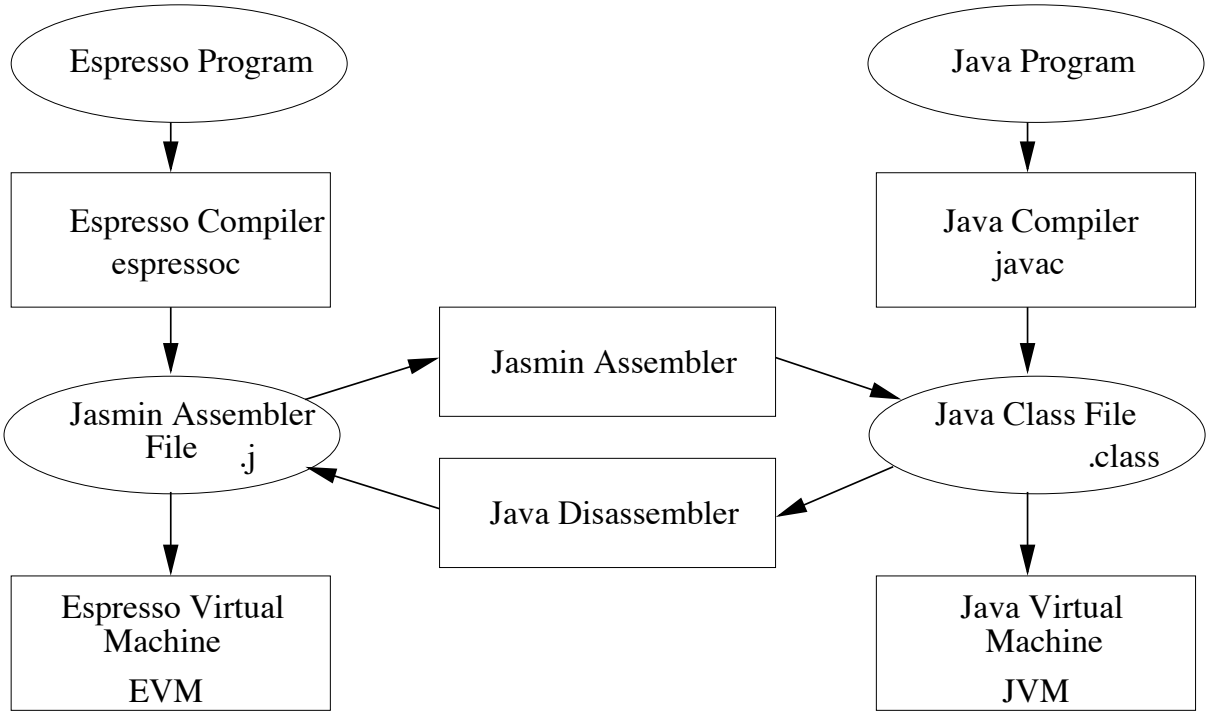
Department of Computer Science
Howard Hughes College of Engineering
University of Nevada, Las Vegas

The Espresso Virtual Machine

The Espresso Virtual Machine is an interpreter for an assembler language which is very closely related to Java Bytecode. The Espresso compiler (the project of the compiler course) outputs Jasmin assembler code, which is the language that you will be developing the Espresso Virtual Machine to interpret/execute.

We will do this in a number of steps, first by building a number of utility classes and then later on by putting it all together into an interpreter.

The Espresso language, the Espresso Compiler, and the Espresso Virtual Machine is in many ways like the Java equivalent. The following figure shows how the different parts compare. It is possible to translate Jasmin files (the output of the Espresso compiler) to runnable Java class files, and it is possible to create files based on Java class files that are almost compatible with Jasmin files (a few differences exist, but since we are not using this feature we do not care about them in this assignment).

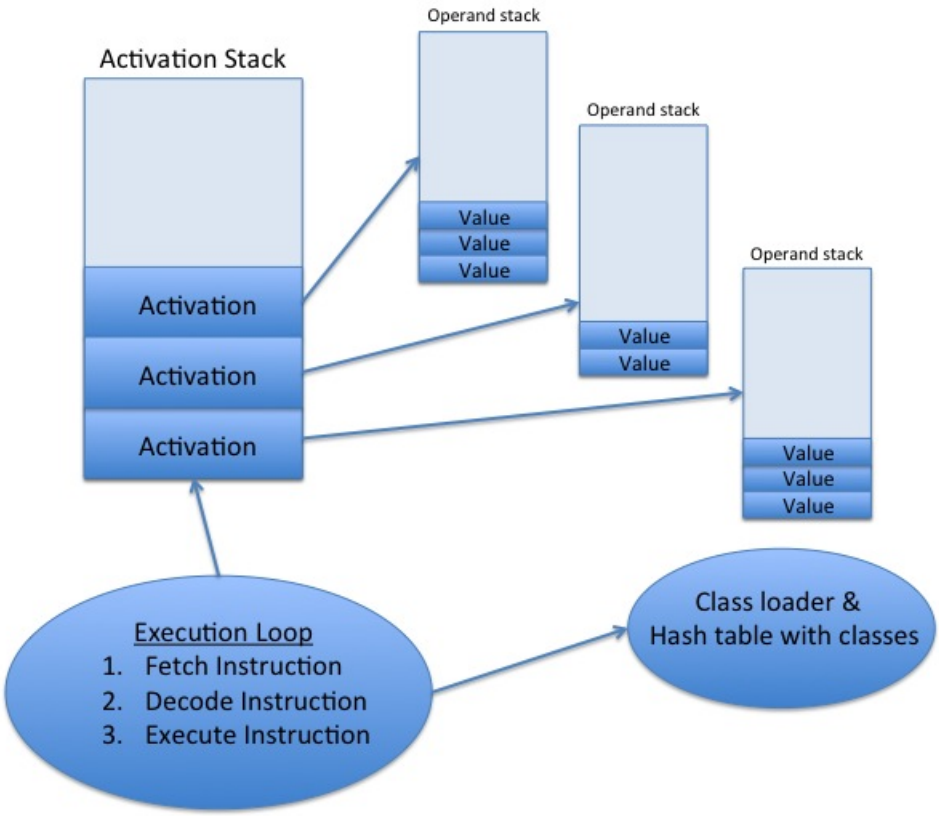


This assignment will concentrate on the EVM part of the above figure.

Before talking more about the EVM it is important to understand that the language (i.e., Jasmin assembler) is closely related to the Espresso language, which is an object oriented language. This of course means that there are instructions in the Jasmin language that support object orientation. Remember the

definition of a class: a collection of data and code, that is, methods and fields. In later phases you will see how we represent a class within the Espresso Virtual Machine. In short this is done through objects created based on a class called Class. More about this later.

The following figure gives a rough overview of the EVM system. It consists of 4 major parts:



- **Class load and class tables** — Classes are loaded as they are needed (i.e., when a class is referenced for the first time, it is loaded by the class loader) and kept in a hash table. Whenever the system needs access to a class it can be looked up here. Classes contain fields and methods, so when code for a method is needed it can be retrieved from a class.

- **Execution loop** — The core of the interpreter is the execution loop. This is basically a large switch statement with a case for each instruction in the EVM instruction set inside a loop that repeats until there is no more code to interpret. This is where you will be doing a lot of work in later phases of the project.

- **The operand stack** — This stack is used for computing. The EVM is a stack based machine, that is, it does not use registers, so all computation must be done using a stack. Each activation has its own operand stack. If the VM is single-threaded, a shared stack is sufficient, but a multi-threaded VM must have a separate stack for each activation.

- **The activation stack** — Since Espresso supports method calls and recursion, we need an activation stack to hold activation records. More about this later as well.

0.1 Espresso and Jasmin Code

As mentioned, we will be interpreting Jasmin assembler code. This code is the output from the Espresso compiler. The Espresso compiler is called `espressoc` and it produces a number of `.j` files, one for each Espresso class. Consider the code on the following page.

The Espresso Compiler is available through the `espressoc` shell script that is included in your handout files.

```
public class Fib {
  public static int fib(int n) {
    if (n <= 1)
      return n;
    else
      return fib(n−1) + fib(n−2);
  }
}
```

The above Espresso code compiles to (Depending on what version of the Espresso Compiler you have):

```
.class public Fib                    ; name of the class
.super java/lang/Object              ; name of its superclass
.method public static  fib(I)I       ; method header
        .limit stack 50              ; max size of the operand stack
        .limit locals 1              ; number of locals (params + variables)

        iload 0                      ; load variable at address 0 onto stack
        iconst_0                     ; load constant 0 onto stack
        if_icmpeq L3                 ; jump to L3 if not equal
        iconst_0                     ; load constant 0 onto stack
        goto L4                      ; jump to L4
L3:     iconst_1                     ; load constant 1 onto stack
L4:     ifeq L1                      ; jump to L1 if top stack elements equal 1
        iconst_1                     ; load constant 1 onto stack
        ireturn                      ; return from this method
        goto L2                      ; jump to L2
L1:     iload 0                      ; load variable at address 0 onto stack
        iconst_1                     ; load constant 1 onto stack
        isub                         ; subtract top 2 stack elements
        invokestatic Fib/fib(I)I     ; invoke method fib() in object of class Fib
        iload 0                      ; load variable at address 0 onto stack
        iconst_2                     ; load constant 2 onto the stack
        isub                         ; subtract top 2 stack elements
        invokestatic Fib/fib(I)I     ; invoke method fib() in object of class Fib
        iadd                         ; add top 2 stack elements
        ireturn                      ; return from this method
L2:     iconst_0                     ; load constant 0 onto the stack
        ireturn                      ; return from this method
.end method                          ; end of method

.method public  <init>()V            ; automatically generated (default) constructor
        .limit stack 50
        .limit locals 1
        aload_0
        invokenonvirtual java/lang/Object/<init>()V
        return
.end method
```

As you can see from the Jasmin assembler code, the class is public, and its name is *Fib*. It subclasses *java/lang/Object* (All classes who do not explicitly specify a superclass always subclass *java/lang/Object*.) The class contains one method called *fib* which takes in one parameter of type *I* (integer) and returns a value of type integer as well. In addition, a default constructor named *<init>* has been provided by the compiler (If no constructor is given, the compiler will automatically generate one for you.)

A number of different classes are used to represent the various Jasmin instructions. More information can be found about these classes by browsing the `Instruction.java` (which will not be available until phase 2) file or consulting the online JavaDoc documentation.

0.2 The Phases of the Project

The project will be in 5 phases, and their focus is as follows:

1. Phase 1: Implementing and testing the operand stack.

2. Phase 2: Implementing and testing a number of operations involving the operand stack.

3. Phase 3: Working with activation records and the execution loops; implementing instructions concerned with moving data from activation records to the operand stack and back. Also branching and loading constants onto the operand stack will be covered.

4. Phase 4: Creating new objects, accessing fields, and invoking methods and constructors.

5. Phase 5: Enabling the virtual machine to execute multi-threaded Espresso programs.

0.3 Obtaining the Handout Code and Compiling the Code

Copy the file `evmu` from the directory `/home/matt/CSC326/EVM` on `java.cs.unlv.edu`. Place this file in the directory where you want to keep the different phases for the project. The command `./evmu install X` where `X` is a phase number between 1 and 5 will install that phase in a directory called `PhaseX`. The command `./evmu isopen X` will determine if phase `X` is open for handout. You can also create the `.tar` file that you need to hand in with the command `./evmu tar`. Should it be necessary to patch an already installed phase, the command `./evmu patch X DIR` will patch phase `X` from an old installation (whenever you install a phase, if that phase already exists, a copy will be made of it). Finally, `./evmu clean X` will delete old installations of phase `X`.

To compile your code, simply type `ant`, and the ant build system will read the `build.xml` file and compile the EVM. If you need to clean your installation, type `ant clean`.

0.4 Handing in the assignment

To hand in the assignment you must execute the `./evmu tar`. The resulting `tar` file is what you must hand in. Hand it in on the `csgfms.cs.unlv.edu` website - one per group is enough, but please fill in the `id.txt` file with the names of the members of the group. I will untar your assignment, type `ant clean` followed by `ant`. This must completely compile your program without errors. If this procedure does **not** work you will not get any points on the phase. **Do not rename any files, especially not the tar file that you create.** Go to `http://csgfms.cs.unlv.edu` to hand in your assignment; you only need to hand one in per group.

0.5 File Structure

The file structure of the EVM system is as follows:

```
EVM/+---- evmu                                  -- install script
    +---- PhaseX/+---- evm                      -- EVM run file
                  +---- build.xml               -- ant bild script
                  +---- evm.info                -- Phase information
                  +---- id.txt                  -- id file - remember to fill ins
                  +---- espressoc               -- espresso compiler
                  +---- Lib/                    -- Library files
                  +---- Include/                -- Include files
                  +---- bin/                    -- the EVM's class files
                  +---- src/+---- EVM.java      -- the EVM driver
                            +---- Activation/
                            +---- EVM/
                            +---- Execution/
                            +---- Instruction/
                            +---- OperandStack/
                            +---- Parser/
                            +---- Scanner/
                            +---- Utilities/
                            +---- Value/
```

Note, not all files or directories will be present until the later phases.

0.6 JavaDoc

I have (started at least) making Java Doc for the project. There should be a `JavaDoc` directory in your EVM directory after installing any phase. Simply open the `index.html` file in any browser and you should have access to the documentation.

Chapter 1

Phase 1 — The Operand Stack

The first part of the assignment involves implementing an operand stack and a number of operations that work on the stack. This stack will later on be used as the operand stack in the Espresso Virtual Machine. Recall the definition of a stack as a LIFO (Last-In-First-Out) structure where elements are 'stacked' on top of each other and only the top most element can be removed. Java does have a *Stack* class, but we will implement our own. The typical operations on a stack are:

- *push*, which places a new value on the top of the stack.
- *pop*, which removes and returns the top element of the stack.
- *top*, which returns the top element without removing it.

Sometimes an operation like *isEmpty*, which returns true if the stack is empty, false otherwise, is available as well.

Since this implementation of a stack is to be used as the operand stack in our EVM, we should design it such that it fits our needs. In our implementation we will call the *top* function *peek* and we will implement it to allow for peeking at any element on the stack without removing it (it will become obvious later on why we need this functionality).

A stack can be implemented in a number of ways, and you are free to do it any which way you like, as long your code satisfies the requirements.

1.1 Stack Values

The values that our stack will hold are objects of the *Value* class. As a matter of fact, they are objects of classes who all subclass the *Value* class, but recall that references to objects of a subclass can be assigned to variables of the superclass, so we will create a stack of *Value* objects, which in time will hold objects of the classes *IntegerValue*, *LongValue*, *FloatValue*, *DoubleValue*, *StringValue*, and *ReferenceValue*. (We do not concern ourselves with objects of type *CharValue*, *ShortValue*, *BooleanValue*, and *ByteValue*, at the moment. We will return to these 4 special values later on, but for now just keep in mind that they will never appear on the stack). The *ReferenceValue* class will not appear until phase 2.

The *Value* class contains a number of parts:

- A number of static final constants starting with *s_* (e.g., *s_integer*). These constants are used to identify the type of the object. Recall that the *Value* class is abstract and can thus not be instantiated (i.e., objects cannot be created based on the *Value* class), but all its subclasses will inherit these constant values along with a *type* field that is set to the correct value when the object is created.

- The definition of the type field (**private final int** *type*). This field is set in the constructor.
- The class' constructor, which takes in the type of the new object and sets the *type* field.
- A method *getType()* that returns the type. This method is necessary as the *type* field is private to the *Value* class.
- A number of static methods that can create various instances of the class' subclasses depending on different values. These include
  - *makeDefaultValueFromSignature(String s)* — Creates a new value object which type is dependent on the value of the parameter *s*. For example, *makeDefaultValueFromSignature("I")* will return an *IntegerValue* object with a default value of 0.
  - *makeValue(Object o)* — Creates a new value object based on the type and value of the object *o*.
  - *makeValue(<type> <p>)* — Like the above method, but one such method has been implemented for each atomic type (integer, long, float, etc.)
  - *signtaure2Type(String s)* — An auxiliary method that will be used later.
  - *type2String()* — An auxiliary method that will be used later.
  - *toString()* — An abstract (i.e., un-implemented) method that all subclasses of *Value* must implement.

Your installation of the EVM assignment should have a `JavaDoc` directory in which you can find more information about the various classes. Note, the JavaDoc is not complete.

Let us briefly look at one of the subclasses of *Value*, *IntegerValue*. The implementation is so short that we can easily show it here:

```
package Value;

public class IntegerValue extends Value {
  private int value;

  public IntegerValue(int iv) {
    super(Value.s integer);
    value = iv;
  }

  public int getValue() {
    return value;
  }

  public String toString() {
    return "" + value;
  }
}
```

As you can see, the *toString()* method has been implemented. A constructor that takes in an integer value has been provided. This constructor stores the integer value with which it was called in a field of type integer, and calls the constructor of the super class (i.e., *Value*) to assure that the type field is set correctly. In addition, an accessor (*getValue()*) has been provided.

All of the other subclasses of *Value* are similar, just with different types. Take a look at them to understand the difference.

All the files are in the *Value* package, so take a closer look at the JavaDoc and the files themselves.

1.2 Stack Specification

Your assignment is to finish the implementation of the operand stack in the `OperandStack.java` file in the `src/OperandStack` directory. It must have the following functionality:

- A constructor that takes in the maximum size that the stack can grow to.
- A push method with the following header: **public void** *push(Value e)*, which places the parameter object *e* on top of the stack. If no space is left on the stack an error message must be displayed and the execution must be terminated (you can use *System.exit(1)* to terminate execution.)
- A pop method with the following header: **public** *Value pop()*, which removes and returns the top element of the stack. If there are no elements on the stack an error must be displayed and execution terminated.
- A peek method with the following header: **public** *Value peek(***int** *n)*, which returns the *n*'th element on the stack without removing it. If there are not enough elements on the stack an error must be displayed and execution terminated.

I have already provided the headers for the methods so you just have to fill them in. For the choice of data structure for the stack itself, an array will do; if you do not want to use an array you will need to rewrite some of the methods at the bottom of the file. An array is a good choice as the Espresso compiler can determine the maximum size of the stack at any given time, so we are never in a situation where we run out of stack space.

1.3 Postfix Notation and Stack Evaluation

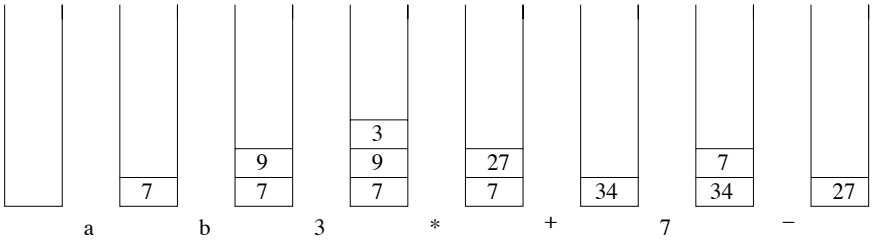
The reason we need this stack is that evaluation of expressions take place on an operand stack. For example, code like $a = b + 3$ will generate class file code that looks like this:

Code for loading b onto the stack
Code for loading 3 onto the stack
Add top two elements onthe stack
Save top stack element into a

Recall that any infix expression like $a + b * 3 - 7$ can be translated into postfix line $a\ b\ 3\ *\ +\ 7\ -$ and evaluated on a stack following these simple rules:

1. If the next token is a variable then push its value onto the stack.
2. If the next token is a constant then push it onto the stack.
3. If the next token is an operator, then pop the top two elements off the stack, apply the operator to them, and push the result back onto the stack.

So if $a = 7$ and $b = 9$, we can evaluate $a + b * 3 - 7 = a\ b\ 3\ *\ +\ 7\ -$ like this:



Since it is the compiler's job to generate the code that represents expressions in postfix notation, all we have to do is interpret/evaluate them, and that is what we need the stack for. Later we will see how instructions like `iconst_X`, `dload`, and `iadd` etc. use the stack to perform calculations.

1.4 Test

You must make your own test programs, of course, but the following little program **must** run correctly.

```
import java.util.*;
import Utilities.Error;
import OperandStack.*;
import Value.*;

public class EVM {
  public static void main(String argv[]) {

    // This is a small example of how to use the stack

    OperandStack operandStack = new OperandStack(100, "OperandStack Test");

    operandStack.push(new IntegerValue(100));
    operandStack.push(Value.makeDefaultValueFromSignature("Z"));
    operandStack.push(Value.makeValue("Hello"));

    System.out.println(operandStack.peek(1));
    System.out.println(operandStack.peek(2));

    // if we have an expression line 1 + 2 we can translate that
    // into postfix like this: 1 2 +
    // so it becomes: push(1), push (2), pop() twice and add and
    // push the result.
    operandStack.push(new IntegerValue(1));
    operandStack.push(new IntegerValue(2));
    Value v1, v2;
    v2 = operandStack.pop();
    v1 = operandStack.pop();
    // Note, since the stack holds 'Value' values we have to cast the
    // values popped to the right kind.
    IntegerValue v3, v4;
    v4 = (IntegerValue)v2;
    v3 = (IntegerValue)v1;

    operandStack.push(new IntegerValue(v3.getValue() + v4.getValue()));
```

```
    System.out.println(operandStack.pop()); // ===> 11
    System.out.println(operandStack.pop()); // ===> Hello
    System.out.println(operandStack.pop()); // ===> true
    System.out.println(operandStack.pop()); // ===> 100
    System.out.println(operandStack.pop()); // ===> Error message & termination
    System.out.println("This line should never be printed out");

    // Remove the code above and replace it by your
    // own tests from the assignment
  }
}
```

The `EVM.java` file in the handout already has the above code.

1.5 Pre-supplied Code

Use the `evmu` script to install the handout code as described in the introduction.

1.6 What to do

Question 1: Implement the operand stack and test that the above program works.

Question 2: Write a test program to compute the following expression using the newly implemented stack: $32 - 7 * (54 + 32)/2$

Remember that *IntegerValues* can be constructed like this:

$$new\ IntegerValue(value);$$

When you write the calls for evaluating the above expression you are really mimicking the operations that the JVM, and later your EVM, will be doing with code generated by a compiler.

Before you start coding, translate the expression to post fix notation; once you have done that, each literal is equal to a push of a new value onto the stack, and each operand is equivalent to two pops, a computation and a push of the result.

1.7 Hand In

Use the command `./evmu tar 1` to create a `.tar` file. Hand in this file without renaming it on `csgfms.cs.unlv.edu` before the deadline.

Chapter 2

Phase 2 — Working With the Operand Stack

The stack we implemented in the first assignment will serve as the operand stack for the Espresso Virtual Machine, which we will implement. This means that all logic and arithmetic operations that the EVM supports must be implemented using this operand stack. However, since you have only implemented a general stack that can push, pop and peek we need to write new methods that can do the logic and arithmetic for us.

The following is a list of the 8 different types of operations we need to perform on elements on the stack:

- **Duplicate** elements on the stack.
- **Convert** elements on the stack from one type to another.
- **Negate** values on the top of the stack.
- **Compare** elements on the top of the stack.
- **Swap** elements on the top of the stack.
- **Logic** operations like `and` and `or`.
- **Arithmetic** operations like + and -.
- **Bit shift** values on the top of the stack.

Keeping in mind that we are implementing all this to eventually work with the EVM we need to take into consideration which instructions of the EVM assembler code we need to support. The file `src/Instruction/RuntimeConstants.java` contains constants for each and every opcode supported by the Java Virtual Machine (JVM). As we are only implementing a subset of these there will be a handful or two that we do not care about.

Each constant is named *opc_XXX* where *XXX* is the actual byte code instruction's name. For example, *opc_aload* represents the *aload* operations (which happens to load a reference value on to the stack).

I have done the *duplicate()* method. All you have to do with this one, is to add it to the execution loop in a later phase. The reason I did this one is that it is a little complicated to get it working correct because in Java **doubles** and **longs** take up two stack entries, but for simplicity, in the EVM they just take up one.

2.1 Binary Operators

Let us start with the binary operators; There are 5 of these: + (`add`), - (`sub`), / (`div`), * (`mul`), and % (`rem`), and these all exist for the four types: **double**, **float**, **long**, and **integer**. This gives us a total of 20 instructions to implement. The table below gives the names of these instructions.

Type	+	-	/	*	%
double	`dadd`	`dsub`	`ddiv`	`dmul`	`drem`
float	`fadd`	`fsub`	`fdiv`	`fmul`	`frem`
long	`ladd`	`lsub`	`ldiv`	`lmul`	`lrem`
integer	`iadd`	`isub`	`idiv`	`imul`	`drem`

As an example, let us look at `dadd`. The two operands that we want to add are currently on the operand stack; these must be popped off the stack, their type checked and finally, the actual **double** values must be added and pushed back on the stack. For all the binary operators we will write a method *binOp* with the following header:

private void *binOp*(**int** *opcode*, **int** *type*, *OperandStack operandStack*)

where *opcode* is one of the constants from `RuntimeConstants.java`, for example *opc_dadd*. *type* is one of the types from the *Value* class, for example *Value.s_double*, and finally *operandStack* is the actual operand stack where the values in question are located.

Remember, the values that are actually on the operand stack are objects of any subclass of class *Value*, so we have to pop them off the stack, and then check that they are of the correct type before casting them and extracting their values. This is the reason we pass in the *type* parameter. Thus, we can begin the implementation of the *binOp()* method like this:

```
Value o1 = operandStack.pop();
Value o2 = operandStack.pop();

// Check that the operands have the right type
if (!(o1.getType() == type && o2.getType() == type))
  Error.error("Error: Type mismatch - operands do not match operator.");

switch (opcode) {
  case RuntimeConstants.opc dadd:
    operandStack.push(new DoubleValue(((DoubleValue)o2).getValue() +
                                      ((DoubleValue)o1).getValue()));
    break;
  case:...
  ...
}
```

As you can see, we first pop off two objects of type *Value*. We cannot be more specific because it could be any subclass of *Value*. We then check the type of the two operands; we check they both have the type corresponding to the parameter *type*. Now we can use the parameter *opCode* to determine which byte code operation we need to perform; this is done using a switch statement. If we are doing a `dadd`, that is, an addition of two double values, the *type* parameter would have the value of *Value.s_double*, and the objects popped off the stack would be checked against this value with respect to their type. Thus, we can cast the *Value* objects to *DoubleValue* objects, and extract their actual *double* values using the *getValue()* method, add them together and create a new double value which we push back on the stack. The semantics for

binOp() can be found in appendix A.1.

Remember, all *Value* objects have a *.getValue()* method that returns their value as the appropriate atomic type. That is, the *.getValue()* of an *IntegerValue* returns an *int*.

2.2 Swapping Stack Elements

Swapping elements on the stack is easy. Simply pop two elements off and push them back on in the opposite order. The only thing you have to keep in mind is that neither of the two elements may be **doubles** or **longs** (This has to do with the fact that **doubles** and **longs** take up 2 spaces on the operand stack in the Java Virtual Machine, but to stay consistent, we keep the rule that these cannot be swapped.) See appendix A.2 for the semantics of the instructions to swap values. Implement your code in a method called *swap(...)*. I have provided the method in the `EVM.java` file; the same goes for all the following methods - you just have to implement the body.

2.3 Negating Values

The 4 numeric value types (**integer**, **long**, **float**, and **double**) can all be negated. Appendix A.3 contains the full specification of the 4 different negation operators: `ineg`, `lneg`, `fneg`, and `dneg`. Implement your code in a method called *negate(...)*.

2.4 Comparing Values

Two values on top of the stack can be compared, and depending on the outcome, a new **integer** value with the values -1, 0, or 1 will appear on the top of the stack. Comparing can only take place between two values of the same type, and only on values of type **long**, **float**, and **double**. Other methods for comparing *integers* and reference values will be considered later. For **floats** and **doubles** we have the following 4 instructions: `fcmpg`, `fcmpl`, `dcmpg`, and `dcmpl`. For our purposes, `fcmpg` and `fcmpl` are identical. The same holds for `dcmpg` and `dcmpl`. For **longs** we have just one instruction: `lcmp`.

To execute these instructions, two values are popped off the stack; their types are verified with respect to the instruction, and -1 is pushed onto the stack if the value of the first is greater than the value of the second object. 0 is pushed onto the stack if the values are the same, and 1 if the second value is greater than the first. See appendix A.4 for the semantics of the instructions. Implement your code in a method called *compare(...)*.

2.5 Converting Values

Only values of type **integer**, **float**, **long**, **double**, and reference (Strings are stored as references to *Java/lang/String*, which is really a reference) can be stored on the stack. However, sometimes we need to convert between these (this happens when we have code like *double a = (double)0;*), or sometimes it happens when we have type coercion, that is, when types are changed automatically by the compiler. The following table shows which conversions are legal in the EVM:

	To						
From	byte	short	char	integer	float	long	double
byte							
short							
char							
integer	`i2b`	`i2s`	`i2c`		`i2f`	`i2l`	`i2d`
float				`f2i`		`f2l`	`f2d`
long				`l2i`	`l2f`		`l2d`
double				`d2i`	`d2f`	`d2l`	

The instructions `i2b`, `i2s`, and `i2c` seem counter-intuitive as only values of **integer**, **long**, **double**, **float** and reference can be on the stack. However, in reality, for these instructions an integer is popped off the stack, the value is cast to either **byte**, **short** or **char** and then pushed back onto the stack as an **integer**. (In effect masking off the lower 8 or 16 bits).

As an example of how to implement these instructions let us consider `l2f`. First, take a look at the header of the *two()* method:

```
public void two(int from, int to, OperandStack operandStack)
```

If we are to execute `l2f` we execute the following call: *two(Value.s_long, Value.s_float, operandStack)*. We need to check that the element on top of the stack is of type *Value.s_long* (or in general, we need to check that the element on the top of the stack is of type *from*) before we proceed. Once we have done this checking we can cast the *Value* object to the correct type (in this situation, a *LongValue*), get the value and cast that to a **float**, and finally create a new *FloatValue* object:

```
public void two(int from, int to, OperandStack operandStack) {
  Value e = operandStack.pop();
  if (e.getType() != from)
    Error.error("OperandStack.two: Type mismatch.");

  switch (from) {
  case Value.s_long:
    long lv = ((LongValue)e).getValue();
    switch (to) {
    case Value.s_float:
      operandStack.push(new FloatValue((float)(lv)));
      break;
    ...
    }
    ...
  }
}
```

Appendix A.5 contains more information on the semantics of the conversion instructions.

2.6 Logic Operations

Exactly three different logic operations exist; these are (bitwise) `and` (&), (bitwise) `or` (|), and (bitwise) `xor` (^). These 3 operations exist for only **integers** (`iand`, `ior`, `ixor`) and **longs** (`land`, `lor`, `lxor`). See appendix A.6 for more information. The code for these operators should be implemented in a method called *logic(...)*.

2.7 Shift Operations

The last type of operations that deal strictly with the operand stack are the shift operations. These can be applied to **integers** and **longs** only, and there are three different ones: shift left, `ishl` and `lshl` (<<), shift right, `ishr`, and `lshl` (>>), and logical shift right, `iushr` and `lushr` (>>>). See appendix A.7 for more information. Place the implementation of the shift operators in the a method called *shift(...)*.

2.8 Pre-supplied Code

As mentioned, the file driver file `EVM.java` contains all the methods you need to implement. Install phase 2 in your EVM directory by typing `./evmu install 2`.

2.9 What to do

Keep in mind that the content of the operand stack can be printed using the method *dump()*.

Question 3: Finish the implementation of the *binOp()* method in the file `EVM.java` and add test cases to the *main()* method to show that your implementation works. Do not forget to include test cases that show that your error checking works as well. In Appendix A.1 you can find the specifications of all the instructions you must implement.

Question 4: Implement the *swap()* method in `EVM.java` and provide test cases for it in the *main()* method. See Appendix A.2 for the semantics of the instruction.

Question 5: Finish the implementation of the *negate()* method in the `EVM.java` file, and provide test cases in *main()* that show that it works. Appendix A.3 contains the details for all the negate instructions.

Question 6: Finish the implementation of the method *cmp()* in `EVM.java` and provide test cases for it in the *main()* method. See Appendix A.4 for details about the instructions.

Question 7: Finish the implementation of *two()* and provide test cases for it in the *main()* method. See Appendix A.5 for implementation details.

Question 8: Finish the implementation of the *logic()* method and provide test cases for it in the *main()* method. Appendix A.6 specifies the implementation details of these instructions.

Question 9: Finish the implementation of the *shift()* method and provide test cases for it in the *main()* method. Appendix A.7 specifies the implementation details of these instructions.

2.10 Hand In

Use the command `./evmu tar 2` to create a `.tar` file. Hand in this file without renaming it on `csgfms.cs.unlv.edu` before the deadline.

Chapter 3

Phase 3 — The Execution Loop & Activation Records

So far we have implemented most of the actual 'computation' that takes place on the stack. We can do arithmetic and logic etc. However, there are still a few things we need to implement that involves the operand stack. We need a number of operations that allow moving values from and to the operand stack. These value can be stored in what we refer to as an activation record. Each time a procedure is called an **activation record** is created. Let us first look at a few instructions concerned with loading constants onto the operand stack and then turn to the heart of the interpreter, the execution loop (The code for the execution loop can be found in `src/Execution/Execution.java`.)

The execution loop contains a large switch statement with one case for each instruction found in `src/Instruction/RuntimeConstants.java`. The implementation of each instruction is added to the switch statement in the execution loops as we progress.

All the methods you implemented in phase 2 in `EVM.java` have been moved to `Execution.java` as this is where we will need them.

3.1 Loading Constants

Constants are often used in Jasmin code. For this reason, a number of instructions for loading constants onto the operand stack are available. Since the numbers -1,0,1,2,3,4,5 (**integers**), as well as the values 0.0 and 1.0 for both **floats** and **doubles**, and a few others are used, often each such constant can be pushed onto the operand stack using special instructions that do not require any operand decoding. For example, the instruction `iconst_m1` pushes the **integer** constant -1 onto the operand stack. The implementation of this instruction is straightforward. We simply add the following line to the switch statement in the execution loop:

```
case RuntimeConstants.opc_iconst_m1:
  operandStack.push(new IntegerValue(-1));
  break;
```

The complete list of instructions for loading constants **without** operand decoding onto the stack are as follows:

Instruction	Explanation	Value Type
`aconst_null`	Loads the reference value null onto the stack.	*ReferenceValue*
`dconst_0`	Loads 0.0 as a **double** onto the stack.	*DoubleValue*
`dconst_1`	Loads 1.0 as a **double** onto the stack.	*DoubleValue*
`fconst_0`	Loads 0.0 as a **float** onto the stack.	*FloatValue*
`fconst_1`	Loads 1.0 as a **float** onto the stack.	*FloatValue*
`fconst_2`	Loads 2.0 as a **float** onto the stack.	*FloatValue*
`lconst_0`	Loads 0 as a **long** onto the stack.	*LongValue*
`lconst_1`	Loads 1 as a **long** onto the stack.	*LongValue*
`iconst_m1`	Loads -1 as a **integer** onto the stack.	*IntegerValue*
`iconst_0`	Loads 0 as a **integer** onto the stack.	*IntegerValue*
`iconst_1`	Loads 1 as a **integer** onto the stack.	*IntegerValue*
`iconst_2`	Loads 2 as a **integer** onto the stack.	*IntegerValue*
`iconst_3`	Loads 3 as a **integer** onto the stack.	*IntegerValue*
`iconst_4`	Loads 4 as a **integer** onto the stack.	*IntegerValue*
`iconst_5`	Loads 5 as a **integer** onto the stack.	*IntegerValue*

The above instructions take care of efficiently loading often used constants onto the stack, but what about constants like 37.98? How do we get such numbers onto the stack?

5 Instructions are available for loading such constants onto the stack: `ldc`, `ldc_w`, `ldc2_w`, `sipush`, `bipush`. The first 3 (all starting with `ldc`) are the same instruction in EVM. All instructions representing one of these 3 can be of one of five different types (depending on the type of the operand). If the operand is an **integer**, the instruction is of *LdcInteger* type. If it is a **float** it is of type *LdcFloat*, if a **string** it is of type *LdcString*, a **long** is of type *LdcLong* and a **double** is of type *LdcDouble*. All these classes are subclasses of the *Ldc* class. See the JavaDocs for more information.

Now might be a good time to study the files in the `Instruction` directory. Remember, the ultimate goal is to evaluate a number of instructions from an object's or class' methods. The Jasmin code associated with a method is stored in an array which we can index using a program counter (pc).

Also remember, not all instructions have the same format. Some have an integer operand, some a float operand, and some have no operands, but in order to be able to store a collection of different instructions in an array we can declared the array as an array of objects of a common super class, i.e.

$$Instruction\ code[\ ] = ...;$$

We can use the accessor method *.getOpCode()* in any subclass of *Instruction* to determine its opcode, which corresponds to the constant in `RuntimeConstants.java`; once we know which instruction we really have, we can cast it to the right one. Spend some time familiarizing yourself with **all** of the files in the `Instruction` directory.

It is worth mentioning that all the Ldc... instructions all implement $getValue()$ which returns a $Value$ object.

The last two instructions for pushing constants onto the operand stack are `bipush` and `sipush`. They both have an integer parameter, which for `bipush` should be cast to a **byte** and then pushed onto the stack as an **integer**. The same holds of `sipush` except it is cast to a **short** before being pushed onto the stack as an **integer**. Finally one last instruction `aconst_null` exists. This instruction pushes a *ReferenceValue* (which we will use to represent a reference to an object in the next phase. The value of a *ReferenceValue* will be a Java reference value that points to an object of type *Object_*, the internal EVM representation of an Espresso object) onto the stack. The value of the object associated with this reference is null. Such a value can be created like this:

```
new ReferenceValue(null)
```

3.2 The Execution Loop

Recall that the Espresso Virtual Machine is an interpreter; it interprets programs written in an assembly like language (Jasmin bytecode). We use a technique of evaluation that is divided into three tempi: Fetch, Decode, and Execute. Fetch refers to getting the next instruction to execute from the code currently being executed. Decode means determining which instruction we have just fetched and it also involves extracting any parameters the instruction might take, and finally, executing the instruction means performing the action that the semantics of the instruction prescribes. Let us look at each of the three phases in turn.

3.2.1 Fetching the Next Instruction

For any execution (except for one exception that we will return to later) we are always executing code within the main method or another method that has been invoked from a different method. The first instruction of a method is located at address 0 of the corresponding code segment, and the following instruction at address 1 and so on. We keep a program counter that keeps track of the address of the next instruction to execute within the method that we are currently executing. We refer to this counter as 'pc'. Fetching the next instruction to execute is done by invoking the method *getInstruction()* on the current method (*currentMethod* is a variable in the execution loop which holds the method currently executing.) The program counter ('pc') can then be incremented by one to point to the following instruction. (We will return to some of the more subtle details of this when we discuss branching and jumping.)

3.2.2 Decoding the Instruction

The *getInstruction()* method returns a value of the *Instruction* class. Such an object knows what operation-code (opcode for short) it is. That is, each instruction knows what kind it is. The opcode can be retrieved by using the method *getOpCode()* on any *Instruction* object. This is the first part of the decoding; We can write code that looks like this:

```
inst = currentMethod.getInstruction(pc);
switch(inst.getOpCode()) {
case RuntimeConstants.opc  aconst null: ...; break;
case RuntimeConstants.opc  aload: . . .; break;
case RuntimeConstants.opc  aload 0: ...; break;
case RuntimeConstants.opc  aload 1: ...; break;
case RuntimeConstants.opc  aload 2: ...; break;
case RuntimeConstants.opc  aload 3: ...; break;
...
}
```

Using the opcode in a switch statement we can write one case for each instruction. As you can see, all the opcode constants are defined in a file called `RuntimeConstants.java`. The second part of decoding is retrieving the parameter associated with the instruction (if the instruction has any parameters). All the various types of instructions that have parameters all have methods for accessing them. You can find these by browsing the files in the `Instruction` directory.
Of the 5 instruction listed in the code above, only *aload* has a parameter. The parameter for *aload* is the address of the object reference within the activation record that is to be loaded onto the operand stack. This address can be extracted (or decoded) from the instruction by using the *getOperand()* method. All instructions with an integer parameter contain the *getOperand()* method. Thus, *inst.getOperand()* retrieves the address of the object reference that is to be loaded onto the stack. If the instruction operand is not a simple integer, then it must be cast to the appropriate instruction subclass before the operand is accessed through a specialized accessor belonging to the subclass.

3.2.3 Executing the Instruction

Once we know which instruction we want to execute and once we have the value of any parameter the instruction might have we can, using the semantic description of the instruction, execute it.

3.3 Activation Records

Recall that each execution of a procedure or a method or an invocation of a method requires that some information about the current context is stored. Why do we need to store information and what kind of information do we need to store? When invoking a method, control is transferred from the invoker to the invokee (i.e., from the method that contains the method invocation to the method being invoked). When the method that is being invoked has finished running, control must be transferred back to the method that made the call, and execution of this method must resume from the instruction immediately following the invocation. Thus, we need to store information about where to return control once the method is finished. Naturally, we cannot simply set aside one variable in our interpreter to contain this information; since we never know how many nested method invocations we might have. So we need to store this information for each call. The preferred way of doing this is to create what is known as an **activation record**, which contains all the needed information. These activation records are stored on a stack, the most recent activation record (on the top of the stack) representing the most recent method invocation. The execution loop has a variable *activation* that should always point to the current activation record. Once a method terminates, it can consult its activation record to retrieve information about the return point in the previous method (i.e., the caller), pop the activation stack (thus removing itself from the current execution) and give control back to the caller (whose activation record now will be the top of the activation stack).

It is not only the return address that is needed in an activation. Consider the following Espresso method:

```
public int sum(int a) {
  int b;
  if (a == 0)
    return 0;
  else {
    b = a + sum(a-1);
    return b;
  }
}
```

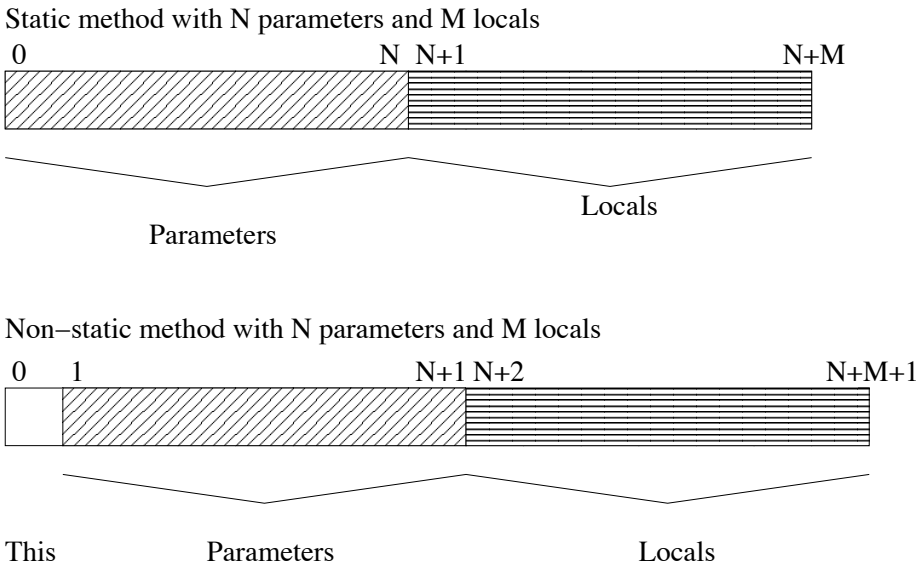
How do we handle the recursive calls to *sum* with respect to the physical location of the parameters and the locals? It it obvious that if we try to statically allocate space for parameters and locals, then recursion cannot be legal; old values will be overwritten with new ones if there is a static location for all parameters and locals. The same holds if space is allocated within the code or in a fixed data segment. The (nicest) way to deal with this problem is of course to allocate space for both locals and parameters for each invocation of a method. The most obvious place to do this is the activation record (since we already have one of them for each method invocation). Thus, each activation record holds the values of the actual parameters (i.e., the values that are passed as arguments to the method invocation) and the local variables declared inside the method.

A few other useful things are stored in the activation record, but you can learn more about that by examining the `Activation/Activation.java` file and reading the corresponding JavaDocs. The activation stack implementation can be seen `Activation/ActivationStack.java`.

One important thing to point out is the layout of the *values* array inside the *Activation* class. This array is used to store both parameters and locals. From the Espresso Virtual Machine's point of view, the is no

difference between the use of a local or a parameter within a method. (This is somewhat of a simplification, but it will work well for EVM). Parameters are stored from address $0..n$ if the method is static, and $1..n+1$ if the method is non-static. The local variables follow the parameters.

Recall that a static method is sometimes referred to as a class method; A static method can be called by *className.mehodName()* as well as *objectRef.mehtodName()*. Thus, a static method cannot rely on having a reference to an actual object available at the time of execution. This is the reason that entry 0 in the *values* array can be utilized for parameters when calling a static method. When calling a non-static method, that is, a method call of the form *objectReference.methodName()* the actual object reference is stored in entry 0. This is commonly referred to as 'this'.



There is however as slight exception to this rule; In Java a main method takes in one argument, which is an array of strings (the command line arguments), and these are placed at address 0 in the activation record for *main()*, thus the local variables start at address 1. Even though we do not have any parameters for the *main()* method, since the Espresso code generation was originally made to run on the Java Virtual Machine, we had to start at address 1 for local variables for the *main()* method; so even though it starts at address 1, it is not because it has a reference to *this* at address 0, it is because this spot is utilized in the JVM. In the EVM we simply ignore address 0 in the *main()* method.

Now, a number of important instructions that connect the operand stack with an activation record exist in the Espresso Virtual Machine. Let us motivate these with a small example. Consider the following Espresso code snippet:

```
public void foo() {
  int a;
  ...
  a = a + 67;
}
```

The value for *a* is stored in the activation record created for the execution of *foo*. *a* is located at address 1 since there are no parameters and since the method is not static. (a reference to the object currently executing *foo* is stored in address 0). In order to evaluate the code associated with the statement *a = a + 67;*, the current value of *a* must be pushed on the operand stack, the value 67 must be pushed on the stack and the `iadd` operation must be executed, leaving the result on top of the stack. This result value must then be stored back into the location where *a* is stored in the *values* array in the activation record.

If you are not 100% sure what the Values array is, go to the file `Activation.java` and find out about it now!

In order to accomplish the transfer of values from an activation record to the operand stack and back we need a number of instructions. These are the load and the store instructions. Such instructions exist for object references (`aload` and `astore`), **doubles** (`dload` and `dstore`), **floats** (`fload` and `fstore`), **integers** (`iload` and `istore`), and **longs** (`lload` and `lstore`). All these instruction have a parameter indicating which address the value is to be loaded from or stored to. In addition to the above mentioned instructions a number of instructions of the form `Xload_Y` and `Xstore_Y`, where `X` is one of `a, d, f, i, l` and `Y` is one of `0, 1, 2` or 3 exist. `Xload_Y` is equivalent to `Xstore Y`. The reason these instructions are in the instruction set of the Java Virtual Machine (and thus also in the Espresso Virtual Machine) is efficiency. When using the form `Xload Y` the value for `Y` must be decoded, and this takes time; when using the form `Xload_Y` the value of `Y` is determined by the opcode, and no decoding of any parameter is necessary. See appendix A.9 for a complete listing and explanation of all the load and store instruction in the EVM.

For completeness, it it worth noting that when a method is invoked, the values of the actual parameters are evaluated and left on the operand stack, and once the method has started running, the first thing the evaluator must do is to remove these values from the operand stack and place them in the appropriate addresses in the activation record. We will return to this when we get to the instructions concerned with invocation of methods.

The only work you need to do for the load and the store instructions is to add each of them to the execution loop of the evaluator, and implement the *load* and *store* methods in `Activation.java`. You will have to do that for each of the instructions from the previous phases as well, but we will return to this task later.

There is one instruction that is related to loading and storing, and that is the `iinc` instruction. The syntax is as follows:

`iinc` *VarAddress Increment*

where *VarAddress* is the address of a local or a parameter in the currently executing method, and *Increment* is the amount this local or parameter is to be increased by. This instruction is of course only valid for **integers**.

I have implemented code for loading classes as they are needed, a so-called class loader. With this code in place you should be able to execute simple Espresso programs that do not make use of fields or method invocations. Such a program could be:

```
public class test {
  public static void main() {
    int a,b,c;
    a = 10;
    b = 90;
    c = a * b + 8;
    a = a + 1;
    b = b * 2;
  }
}
```

which when compiled to Jasmin assembler code looks like:

```
.class public test
.super java/lang/Object
```

```
.method public static main()V
        .limit stack 50
        .limit locals 4

        bipush 10
        istore_1
        bipush 90
        istore_2
        iload_1
        iload_2
        imul
        bipush 8
        iadd
        istore_3
        iload_1
        iconst_1
        iadd
        istore_1
        iload_2
        iconst_2
        imul
        istore_2
     return
.end method
```

When you execute the EVM with the Jasmin file obtained from compiling the above program, the first thing the evaluator will do is to search the class for a '**public static** *void main()*' method and start execution from its first instruction.

Since the *main()* method is static, no object is required. Just like Java, no initial object is created. The only way an object can be created is by using the `new` instruction. We return to this in the next phase.

3.4 Branching

So far we have not considered code that branches. Consider the following code:

```
...
if (a == 0)
  b = 10;
else
  b = 99;
```

This code could produce Jasmin code like this (I have optimized it a little compared to what the compiler actually produces.):

```
        iload 1
        iconst_0
        if_icmpeq L4
        goto L4
L4:     bipush 10
```

```
        istore 2
        goto L2
L1:     bipush 99
        istore 2
L2:     nop
```

As you can see a number of different jumps and branches exist. First the unconditional jump/branch of the `goto` or `goto_w` instruction. When such an instruction is reached *pc* is set to the address where the label referred to by the instruction is located. The target for any jump or branch can only be within the method. The instruction generated for a *goto* is of type *Jump* (See `JumpInstruction.java`). This instruction object contains a method *getLabelAddress()* which returns the address of which ever label is passed to it. This is useful for determining the new value of *pc*. Two other types of branching exist. The first, and simplest, is integer value branching. Depending on the value of the element on top of the stack, *pc* is either incremented by one (no branching takes place) or set to the address of the label referred to in the instruction. Here is a complete list of the integer branching instructions:

Instruction	Explanation
`ifeq L`	Sets *pc* to the address of label `L` if the value of the element on the top of the stack is 0.
`ifne L`	Sets *pc* to the address of label `L` if the value of the element on the top of the stack is not 0.
`iflt L`	Sets *pc* to the address of label `L` if the value of the element on the top of the stack less than 0.
`ifle L`	Sets *pc* to the address of label `L` if the value of the element on the top of the stack is less than or equal to 0.
`ifgt L`	Sets *pc* to the address of label `L` if the value of the element on the top of the stack greater than 0.
`ifge L`	Sets *pc* to the address of label `L` if the value of the element on the top of the stack greater than or equal to 0.

All these instructions should be implemented in a method called *ifBranch()*. Remember to check that the value on top of the stack is actually an integer and not something else.

Remember when we looked at comparing values, we did not implement any instructions for comparing **integers** that left a value of -1, 0, or 1 on the stack. Such instructions do not exist for **integers**. Instead, instructions exist for comparing **and** branching at the same time. Here is the list:

Instruction	Explanation
`if_icmpeq L`	Compares the top two **integer** values on the stack and sets *pc* to the address of label `L` if they have the same value, or increases *pc* by one if not.
`if_icmpne L`	Compares the top two **integer** values on the stack and sets *pc* to the address of label `L` if they do not have the same value, or increases *pc* by one if they do.
`if_icmplt L`	Compares the top two **integer** values on the stack and sets *pc* to the address of label `L` if the top element is larger than the second element, or increases *pc* by one if not.
`if_icmple L`	Compares the top two **integer** values on the stack and sets *pc* to the address of label `L` if the top element is larger or equal to the second element, or increases *pc* by one if not.
`if_icmpgt L`	Compares the top two **integer** values on the stack and sets *pc* to the address of label `L` if the top element is smaller than the second element, or increases *pc* by one if not.
`if_icmpge L`	Compares the top two **integer** values on the stack and sets *pc* to the address of label `L` if the top element is smaller or equal to the second element, or increases *pc* by one if not.

Two instructions like the ones above exist for reference values as well:

Instruction	Explanation
`if_acmpeq L`	Compares the top two (reference) values on the stack and sets *pc* to the address of label `L` if the two values represent the same object, that is, if they are the same.
`if_acmpeq L`	Compares the top two (reference) values on the stack and sets *pc* to the address of label `L` if the two values do not represent the same object, that is, if they are not the same.

All these branching instructions are implemented in a method named *ifcmpBranch()*.

In addition, two other comparing instructions exist for reference:

Instruction	Explanation
`ifnull L`	Sets *pc* to the address of label `L` if the value on the top of the stack is an object of type *ReferenceValue* whose value is null.
`ifnonnull L`	Sets *pc* to the address of label `L` if the value on the top of the stack is an object of type *ReferenceValue* whose value is not null.

3.5 Pre-supplied Code

As usual, I have provided some code; install it using the command `./evmu install 3`.

3.6 What to do

Question 12: Finish the implementation of *loadConstant* and In addition implement the `pop` and `pop2` instructions. Both pop instructions remove the top value of the stack.

Question 13: Implement *load()* and *store()* in `Activation.java` and Also implement *iinc ()*.

Question 14: Implement the branching instructions in *ifBranch()* and *ifCmpBranch()* and add them to the execution loop. Appendix A.10 contains the semantics for the instructions. Keep in mind that the *break* instruction that you normally put at the end of each case in the execution loop should be a *continue* for all the branching instructions. The reason for this is that *pc* is incremented each time we break out of the case statement in the execution loop, and when we want to continue execution from a specific address, we don't want one added to it. If we did, we would end up executing the instruction not immediately following the label, but the one following the label.

Question 15: Test that your evaluator works so far. Add a call to *activation.dump()* at the end of the execution loop to allow inspection of the variables to see if they contain the correct values. You can also use *operandStack.dump()* to inspect the content of the operand stack. Also you can consult the generated `EVM-1.log` file

3.7 Hand In

Use the command `./evmu tar 3` to create a `.tar` file. Hand in this file without renaming it on `csgfms.cs.unlv.edu` before the deadline.

Chapter 4

Phase 4 — Creating Objects, Accessing Fields and Invoking Methods

When a class is loaded, the first thing the class loader does is to look for any static initializer in the class. A static initializer is a piece of code that looks like **static** { ... }. A static initializer is a method named `<clinit>`. If such a method exists, it is executed **once** when the class is loaded. Look at the code found in *ClassList.java* to understand how this happens. Note, this is the only time a new execution is created. Since the Espresso Virtual Machine is single threaded, one execution is enough. However, since classes are loaded on the fly, that is, a class is not loaded until the first time it is referenced, if such a class has a static initializer, the easiest thing to do is to simply create a new temporary execution to execute the code in the static initializer. Once this execution terminates normal code execution is resumed. This matters when evaluating a return instruction in an execution of a static initializer.

4.1 Creating New Objects

Objects are created based on classes. In order to create an object the instruction `new` must be used. It is common to think of an object as a collection of data and methods (i.e., fields and methods). However, for obvious reasons, each object does not contain a copy of the methods from the class. These are stored in the class, and each object has access to them. However, all non-static fields are stored in the object. The object class in EVM is called *Object_* (We cannot use the name *Object* as that is reserved in Java.) An object of the *Object_* class is the actual value of the *ReferenceValue*. The *Object_* class looks like this (implementation details of some of the methods has been omitted, see the file `src/EVM/Object_.java` for details):

```
package EVM;
import java.util.*;
import Value.*;
import Instruction.FieldRef;
import Utilities.Error;

public class Object {
  private Class thisClass;
  public Object superClassObj;
  public Hashtable fieldValues;
```

```
  // An object stores all its field values in a hashtable as pairs
  // (field-name,value)= (name:string,value:Fieldvalue(Value,Field))
  // All static fields are stored in the class and not in the object.
  public Object (Class thisClass) {
    this.thisClass = thisClass;
    fieldValues = new Hashtable();

    if (thisClass.hasSuperClass())
      superClassObj = new Object (ClassList.getClass(thisClass.getSuperClass()));

    for(Enumeration e=thisClass.getFields().elements(); e.hasMoreElements(); ) {
      // insert into the hash table with the name of the field a fieldvalue:
      Field f = (Field)e.nextElement();
      FieldValue fv;
      // If the field is static then it does not belong to the object.
      if (!f.isStatic()) {
        fv = new FieldValue(Value.makeDefaultValueFromSignature(f.getSignature()), f);
        fieldValues.put(f.getName(), fv);
      }
    }
  }
  ...
}
```

As you can see, each object of the *Object_* class contains a reference to the class it was created from, a hash table with all non-static fields, and a reference to any object representing a super class this object might have. The 'super object' is initialized before anything else (this is obvious from the constructor above). The syntax of the `new` instruction is:

`new` *class-name*

The `new` instruction must leave a reference to the newly created object on the operand stack (i.e., a value of type *ReferenceValue*, whose internal value is a reference to an object of the *Object_* class.)

As you recall, when an object is created in Java, the call to *new* will cause some constructor to be executed (which one depends on the number and types of the parameters passed to *new*). However, in Jasmin assembler, it is not the job of the `new` instruction to call the constructor. This is done explicitly in the following instructions. We will return to this when we talk about invoking methods.

Two important instructions that have to do with objects exist. The first is the `instanceof` instruction. This instruction checks if the value on top of the stack is a reference value whose value represents an object of whichever class is given as parameter to `instanceof`. That is, for `instanceof A`, we must check that the value on top of the stack is a reference value whose *Object_* value represents an EVM object based on the class `A`. If it is, it leaves the **integer** value 1 on top of the stack. If it is not, the value 0 is pushed on the stack, and if the value of the reference object is null, the value 0 is also left on the stack. The implementation is not at simple as you might think!

The other instruction is the `checkcast`, which checks that the value on top of the stack is a reference value which represents an object which can be cast to whichever class is given as parameter to `checkcast`. If this is not the case, execution is terminated with a 'class cast error' error message. See appendix A.11 for syntax and semantics for these two instructions.

The major difference between `instanceof` and `checkcast` is that `instanceof` removes the reference

value on the stack and replaces it with an **integer** value. `checkcast` does not remove the reference value and pushes nothing onto the operand stack either.

4.2 Fields

A field is defined in the Jasmin assembler file with the following syntax:

`.field` *modifiers name signature* [= *value*]

modifiers is as usual either `public`, `private` and/or `static`. *name* is the name of the field, and *signature* is a representation of the type of the field. *value* is a constant value that is assigned to the field when the object is created (if the field is non-static) or when the class is loaded (if the field is static). Examples are:

```
.field private static final Q I = 44488
.field private static final R D = 3399.00
.field public seed I
.field private position I
```

This table summarizes the various possibilities for signatures:

Type Name	Type Signature
boolean	Z
byte	B
short	S
char	C
integer	I
long	L
float	F
double	D
String	Ljava/Lang/String;
void	V
class type 'myClass'	LmyClass;

Note, values of type void cannot be created.

Fields are stored in 2 different places: static fields, or class fields, are associated with the class and not objects created based on the class. Only **one** instance of static fields exist, and their values are stored in the class. Any initialization of such fields take place when the class is loaded.

Non-static fields belong in objects; each time an object is created, non-static fields are instantiated inside the object. See `Object_.java` for more details.
To access static fields we use the instructions `putstatic` and `getstatic`. Their syntax is:

`getstatic` *class-name/field-name signature*
`putstatic` *class-name/field-name signature*

and they generate an instruction of type *FieldRef* (See `FieldRef.java`). `getstatic` will leave the value of the field on top of the operand stack, and `putstatic` will require the value to be assigned to the field to be on the top of the operand stack.

To access non-static fields we use the instructions `putfield` and `getfield`. In addition, a reference to the object in which the field is located must be on the top of the operand stack. For `putfield` the value to be assigned to the field must also be on the operand stack, and of `getfield` the field's value will be copied

to the operand stack after the object reference has been removed. The syntax for these two instructions is:

`getfield` *class-name/field-name signature*

`putfield` *class-name/field-name signature*

Keep in mind that it is the compiler's job to assure that the correct instructions are generated, but for robustness we should do as much checking as possible to assure that illegal instructions do not crash the EVM. Examples of putting and getting values from fields can be:

```
getstatic Main/courseLength I
getstatic Main/hareChar C
getfield Random/seed I
putfield Hare/position I
putfield Tortoise/position I
```

Appendix A.12 contains the syntax and semantics for the field manipulation instructions.

4.3 Constructors and Methods

We need to implement a couple of instructions associated with method and constructor calls. There are four different invocation instructions: `invokevirtual`, `invokenonvirtual`, and `invokestatic` all of which will be of type *MethodInvocation*, and `invokeinterface` which will be of type *InterfaceInvocation*.

Their syntax are:

`invokevirtual` *class-name/method-name*(signature)return-type
`invokenonvirtual` *class-name/method-name*(signature)return-type
`invokestatic` *class-name/method-name*(signature)return-type
`invokeinterface` *class-name/method-name*(signature)return-type parameter-count

`invokevirtual` and `invokenonvirtual` are similar in implementation. The only difference is in the way the method is looked up in the class. For `invokenonvirtual` the method is looked up in the class specified in the instruction (and only in that class). For `invokevirtual` the method is looked up in the class that corresponds to the object that is on the stack, if not found there, then it is looked up in the class' super class etc. This means that we need to use the *peek()* method on the operand stack to get to the object reference the method is being invoked on. Once we have it, we can find the correct class. This is similar to the way `invokeinterface` is handled. You can use the methods *getNonVirtualMethod()* and *getVirtualMethod()* inside *Class* to find the correct method.

The class and the method name can be extracted from the instruction which is of type *MethodInvocation*. Once you have the class name you can look up the class in the class list (you must add "/" + *mi.signature()* because as Espresso allows method overloading the name is not enough to distinguish the methods from each other), then you can use this class to get the method in question. Once the method has been found a new activation should be created (the operand stack should contain a reference to the object that the method is being invoked on, if it is non-static, and the parameters that are being passed to the method), it should be pushed on the activation stack, and *pc* and *currentMethod* should be set correctly. Finally *this_* should be updated to point to the object that contains the method currently being executed. Do not forget to update the *activation* variable.

Keep in mind that the operand stack contains a reference to the object on which the method is invoked as well as all the values of the actual parameters. Both the object reference and the parameters are removed in the creation of the new activation record.

Constructors are treated as methods. All constructors are named *<init>*, and are always invoked using `invokenonvirtual`

Methods can be either static or non-static (constructors are always non-static). Non-static methods require a reference to the object on which the method is being invoked to be on the operand stack as well as any actual parameters. The `invokeinterface` is similar; for execution purposes it is similar to `invokevirtual`. Since we know that any method that implements an interface must implement all methods of the interface, and since we also know that if any class subclasses another class that implements an interface then all methods in the interface will be implemented somewhere in the class hierarchy, we can look for the method like we look for a regular virtual method. However, there is one little catch. Since the `invokeinterface` instruction does not know what type of object it is being invoked on (i.e., what class the actual object is an instance of) we cannot use the name of the interface as the class name; we have to look at the object on the operand stack (under the parameters), get the class name from it and then look for the method to invoke. Here the *peek()* operation on the operand stack comes in handy. We know that it is located just below the actual parameters, so by looking at the number of parameters (*getParamCount()*) we can easily find the object reference we are looking for. This is the only difference between `invokeinterface` and `invokevirtual`.

Because of various reasons that will be clear when you take the compiler course we have had to make a few 'hacks' to get things running. One of these is that all calls to methods in a class named *java/lang/Object* should be ignored (so for `invokenonvirtual`, if the class name is *java/lang/Object* just pop the stack and break), and all invocations of methods in a class called *Io* should be take care of in a specific way. I have implemented the *Io* calls, so you just have to call my code. That is, if the class name is `Io`, call the *doIO()* method and then break.

Any call to *System.out.print* or *System.out.println* (remember to import *System*) will generate calls to *Io*.

When a method terminates it must reach a return instruction. It is not legal to have methods that are not terminated by return instructions. Depending on the return type of the method, a number of different return instructions are available: `dreturn`, `freturn`, `areturn`, `ireturn`, `lreturn`, and `return`. If a return statement is reached in the *main* method there is nothing to do but terminate the execution loop. If the method we are returning from is a static initializer we need to pop the activation stack and terminate the execution loops (Remember, we create a new execution for static initializers.), and for regular methods, we need to find the return address of the caller (this is stored in the activation), we need to pop the activation stack and set *currentMethod* and *activation* and *this_*. Like the jump instructions, instead of breaking, we continue after each of the returns.

4.4 Pre-supplied Code

Install ithe handout code using the command `./evmu install 4`.

4.5 What to do

Question 16: Implement `new` by adding it to the execution loop. You won't be able to test it with the output from the Espresso compiler directly as it automatically generates calls to a constructor, but you can edit the `.j` file and simply remove the constructor call or just ignore it.

Question 17: Implement `putstatic`, `putfield`, `getstatic`, and `getfield`. Appendix A.12 contains the semantics.

Question 18: Implement the 4 invocation instructions. Their semantics can be found in Appendix A.13.

Question 19: Implement the return instructions, `instanceof`, and `checkcast`.

This completes the implementation of the Espresso Virtual Machine. This means that your EVM should be able to correctly interpret/execute the output from the Espresso compiler. You should have a file called `espressoc` which invokes the Espresso compiler and generates a number of `.j` files. A number of Espresso files can be found in in the `Espresso` sub-directory of your EVM directory. You should be able to run all these files. See the source files for which type of input you need to provide when you run them.

4.6 Hand In

Use the command `./evmu tar 4` to create a `.tar` file. Hand in this file without renaming it on `csgfms.cs.unlv.edu` before the deadline.

Chapter 5

Phase 5 - Multi-threading in the EVM

The current implementation of the Espresso Virtual Machine is single threaded; that is, everything is done in a single thread of control. It is not possible to execute for example two different methods at the same time. However, most languages do have some way of executing multi-threaded programs.

Multi-threading can be a complicated business, but let us look at the basics of multi-threading in Java. A new thread can be started, and it will begin executing independently of the thread that started it, resulting in two threads running at the same time. In Java there are two basic ways to create and run threads. The first is by extending the *Thread* class, and the second is to implement the *Runnable* interface. Let us briefly look at both approaches.

5.1 Extending the *Thread* Class

Any class that extends the *Thread* class should implement the **public void** *run*() { ... } method. This method is the one that will get executed when the thread is started (a thread can be started by invoking the *start()* method on it). Let us start with declaring a class that we can use for producing some output. I have placed some code in class *T* to slow down the execution so that we can see the concurrent execution later on.

```
public class T {
  public void foo(String s) {
    for (int i=0; i<5; i++) {
      try {
        Thread.sleep(10);
      } catch (Exception e) { } // Sleep for a while
      System.out.println(s);
    }
  }
}
```

Let us now implement a class *T1* that extends *Thread*:

```
public class T1 extends Thread {
  public void run() {
    new T().foo("Hello from the Thread in T1");
  }
}
```

To start a new thread based on the *T1* class simply create an instance of it and invoke the *start()* method:

```
new T1().start();
```

The following code starts a new *T1* thread and continues running more code:

```
public class Tr {
  public static void main(String args[]) {
    new T1().start();
    for (int i=0; i<5; i++) {
      try {
        Thread.sleep(10);
      } catch (Exception e) { }
             System.out.println("Hello from the Main method.");
    }
  }
}
```

Here is an example of running the *Tr* class:

```
[matt@Air] java Tr
Hello from the Thread in T1
Hello from the Main method
Hello from the Main method
Hello from the Thread in T1
Hello from the Main method
Hello from the Thread in T1
Hello from the Thread in T1
Hello from the Main method
Hello from the Main method
Hello from the Thread in T1
```

We actually do not need to define a new class that extends *Thread*. We can simply extend in on the fly in the following way:

```
new Thread(){
  public void run() {
    newT().foo("Hello from the anonymous Thread.");
  }
}.start();
```

This means we could have written the code in the previous example in the following way:

```
public static void main(String args[]) {
  new Thread(){
    public void run() {
      newT().foo("Hello from the anonymous Thread.");
    }
  }.start();
  for (int i=0; i<5; i++) {
    try {
      Thread.sleep(10);
    } catch (Exception e) { }
  System.out.println("Hello from the Main method."); }
}
```

Let us look at how to accomplish the same using the second technique.

5.2 Implementing the *Runnable* Interface

The *Runnable* interface has just one method, namely the abstract method *run*, which we must reimplement.

```
public class T2 implements Runnable {
  public void run() {
    new T().foo("Hello from the Thread in T2");
  }
}
```

An instance of *T2* can be passed as a parameter to the constructor of a *Thread* and subsequently the thread can be started:

```
public static void main(String args[]) {
  T2 t2 = new T2();
  new Thread(t2).start();
  for (int i=0; i<5; i++) {
    try {
      Thread.sleep(10);
    } catch (Exception e) { }
    System.out.println("Hello from the Main method.");
  }
}
```

The first two lines of *main* could have been written as

```
new Thread(new T2()).start();
```

Java has lots of other classes and methods that are concerned with multi-threading, and we will return to two of them (*wait*() and *notify*()) a little later.

It should be noted, when using an anonymous class for creating a new thread, any variable declared outside this anonymous class and referenced inside it must be declared **final**, that is made a constant.

5.3 Requirements For a Multi-threaded EVM

Before we start changing the code, we need to consider a couple of things.

- Which of the ways to start a new thread should we use? To keep the amount of code to a minimum (remember, I said we could do this in 20 lines of code), we will make use of the anonymous class approach.
- What exactly do we want to run in the thread? Basically we want to run an instance of the *Execution* in a new thread.
- What determines to start a new thread? For this discussion, see the next section.
- Recall that the EVM prints out some nice statistics at the end. To assure that only the first thread (the main one!) prints this, and to assure that it does not print it before every other thread is done, we need some mechanism to hold back the main thread from terminating before all the other threads have terminated. For how to do this see section 5.5.
- Which EVM instructions do we have to make changes to? This is simple: `invokevirtual` and `return`.

5.4 What Starts a New Thread

Since Espresso is very close to Java, we should try to mimic what Java does. However, Espresso does not allow anonymous classes, so that approach is out. An easy approach (that does not even require any changes to the Espresso compiler), is to include a class like this:

```
public class Thread {
  public Thread() { }
  public void run() { }
  public final void start() {
    this.run(); }
}
```

and then implement the EVM to recognize invocations of a *start()* method on an object whose class extends *Thread* (or whose super class or super class' super class etc. extends *Thread*), in other words, if the *start()* method is invoked on an object which has *Thread* somewhere in the class hierarchy, then we should create a new Java thread and run an instance of an *Execution* for this object.

The `Include` directory contains a file called `Thread.java` with the above content. This is needed for the Espresso compiler.

Let us look at a small multi-threaded Espresso program using this technique (Let us assume that *Thread* is defined as above):

```
import System;
import Thread;

public class MyThread extends Thread {
  public void run() {
    for (int i=0;i<5;i++)
      System.out.println("Hello from Thread");
  }
}

public class ThreadTest {
  public static void main() {
    new MyThread().start();
    for (int i=0;i<5;i++)
      System.out.println("Hello from Main");
  }
}
```

Let us look at the Jasmin code that the compiler creates for the *ThreadTest* class:

```
.method public static main()V
      .limit stack 50
      .limit locals 2

      new MyThread
      dup
      invokespecial MyThread/<init>()V
      invokevirtual MyThread/start()V
      iconst_0
      istore_1
L1:   iload_1
```

```
      iconst_5
      if_icmplt L4
      iconst_0
      goto L5
L4:   iconst_1
L5:   ifeq L2
      getstatic System/out LIo;
      pop
      ldc "Hello from Main"
      invokestatic Io/println(Ljava/lang/String;)V
L3:   iload_1
      iinc 1 1
      pop
      goto L1
L2:   return
.end method
```

As we can see, the *start()* method is invoked in the line `invokevirtual MyThread/start()V`, so in the implementation of `invokevirtual` we can create a new *Execution* :

```
final Execution e = new Execution("Thread","start/()V", operandStack);
```

and execute it: *e.execute*(*trace*) inside in a new anonymous Java thread as we discussed in the previous section. Remember, the thread that starts the new thread must continue with the next instruction, so we don't need to do anything to *pc*, just do a **break**, and *pc* will be incremented by 1 automatically.

Now, what do we have to do in the `return` case? Right now we have something like this:

```
if (currentMethod.getMethodName().equals("main")) {
  done = true;
}
```

We can use the *activation.getReturnAddress()* to determine if we are executing a method which is not going to return to a caller. If the return value of *getReturnAddress()* is -1, then we are in one of two situations:

- We are in the **public static void** *main*() method; the method that is loaded and executed when the EVM starts.
- We are in the run method of a class that extends *Thread*.

In both situations, if *activation.getReturnAddress()* equals -1, we want to set *done* to **true** so that the execution loop terminates. Do not forget to close the *PrintWriter* (*pw.close()*).

This should do the trick if we do not care about printing the statistics. In the single threaded EVM, the statistics are printed at the bottom of the execution loop, that is, when it terminates, but that will not work for the multi-threaded EVM as several instances of the execution loop will terminate, so if we want to print the statistics, we have to do it in the `opc_return` case but only if we are returning from a method called `main` that is static and has signature `"()V"`.

5.5 Waiting for Threads to Terminate

Consider this execution of our little test program:

```
Loading class : ThreadTest...Done!
Loading class : MyThread...Done!
Loading class : Thread...Done!
Hello from Thread
Hello from Main
Hello from Main
Hello from Main
Hello from Thread
Hello from Main
Hello from Thread
Hello from Main
Hello from Thread
Number of instructions executed: 359
Number of method invocations...: 30
Number of threads executed.....: 2
Hello from Thread
```

See what happened? The last `"Hello from Thread"` was printed **after** the statistics, which might be wrong anyway because we still have threads running. How do we prevent this from happening? A simple (theoretical) solution is to assure that the main thread (the one started by the EVM) never terminates and prints the statistics **before** all the other threads have terminated. Here is a possible approach:

- Add a variable *threadCount* that counts how many new threads are currently running.
- Before creating a new *Execution* and starting the new thread, increment this counter.
- Before a thread terminates, decrement the counter.
- Before printing the statistics, wait for this counter to become 0.

Such a counter must naturally be shared between all the threads, they all need to either read and/or write it. This could potentially be a big problem; imagine what can happen when more than one thread has access to a shared variable. Two threads could write it at the same time etc. We call this a *race condition*, and we must avoid such a thing.
Luckily Java can help us here in the follow way. Let us write a new class called *ThreadCounter* which holds the counter and a couple of methods for incrementing, decrementing, and accessing the counter:

```
public class ThreadCounter {
  private int counter = 0;

  public synchronized void increaseThreadCount() {
    counter++;
  }

  public synchronized void decreaseThreadCount() {
    counter--;
    // here goes some magic
  }

  public synchronized int getThreadCount() {
    return counter;
  }
}
```

If you look closely at this code, you will see a new modifier keyword: **synchronized**. Any synchronized Java method of an object may never execute while another synchronized method of the same object is being executed. By declaring all the methods in this class synchronized, we can assure safe and race-condition free access to the counter variable (the *counter* field). So all we need to do is to create an instance of *ThreadCounter*, and store that somewhere where every thread can access it; An obvious place is as a static field in `Activation.java`: **public static** *ThreadCounter threadCounter* = **new** *ThreadCounter*();

Now before we create the next *Execution*, we can increment the number of new threads by *Activation.threadCounter.incrementThreadCounter*();, and in the new thread, immediately after the *e.execute*(*trace*), we can decrement the thread counter by a call to *decrementThreadCounter*().

We could now put code like this in the return case before printing the statistics:

```
while (Activation.threadCounter.getThreadCount() > 0)
 ;
```

In other words, we are doing a busy wait, spinning and wasting cycles in the JVM. Naturally, this is **not** the way to do it! What we want to do is to put the main thread to sleep, and have someone wake it back up when the *count* field in the *ThreadCounter* object reaches 0. (this is the "// here goes some magic" part in the *decrementThreadCount*().)

Java has a method to accomplish just this! A thread can be put to sleep by calling *wait*() on any object. The same thread can be woken back up by some other thread calling *notify*() or *notifyAll*() on the same object. *notifyAll*() is used if more than one thread might be waiting on the same object.

Now, what object should we use? How about *Activation.threadCounter*? So, the main method should call *Activation.threadCounter.wait*() if the *Activation.threadCounter.getThreadCount*() is greater than 0. The actual code looks like this:

```
try {
  synchronized (Activation.threadCounter) {
    while (Activation.threadCounter.getThreadCount() > 0)
      Activation.threadCounter.wait();
  }
} catch (java.lang.InterruptedException ie) {
  System.out.println("Waiting for thread counter to become 0 failed!");
  System.exit(1);
}
```

Java requires that we wrap the **synchronized** block around the *wait()* call, and by wrapping it around the **while** statements as well, we avoid a very subtle error that we will return to a little later.
All we need to do now is fill the the magic in the *ThreadCounter*'s *deacreaseThreadCounter*() method. If the counter reaches 0, that means we are about to terminate the last created thread. All we have to do is issue a *notify*() on the *Activation.threadCounter* object:

```
if (counter == 0)
  this.notify();
```

Alternatively, we could have used the *Thread* class' *join()* method, which allows any thread to wait for another thread to die, but what we have here will work just fine for us.

5.6 A Subtle Error

As described earlier, it is important that the **while** statement be inside the synchronized block. The code in the block grabs a lock on the *Activation.threadCounter* object, which means that no other method may execute any method on that object while the lock is held. If we did not do this, we can imagine that the

while test succeeds (because *counter* > 0). Now imagine that the thread that is executing this code is swapped out and another thread is swapped in; if this other thread decrements the counter to 0 and issues the notify, no one is waiting yet, and the notify is lost. When the first thread is resumed again, it will execute the *wait*(), and be stuck there forever! Therefore, the **while** must be *inside* the synchronized block.

5.7 Multi Threading with Runnables in Espresso

The approach described above relies on extending the *Thread* class. Recall the implementation of the *start()* method: It simply executed the *run()* method, and since this is done virtually, the extending class (the subclass) will have code to run in its *run()* method. It should be noted that the file `Thread.j` in the `Lib` folder **actually** contains code that is run by the EVM, namely the aforementioned *start()* method. The `Thread.j` file found in the `Lib` directory is produced by the Espresso compiler.

Now, if we want to mimic Java by allowing threads to be created based on a *Runnable* interface, then we need to make some small changes. First off, we need to write a *Runnable* interface:

```
public interface Runnable {
  public void run() ;
}
```

Since *Runnable* is an interface, there will be no code to run, However, for the compiler to function correctly, it must know about this file, so place it as it appears above in a file called *Runnable.java* in the `Include` directory. Now when you want to use the *Runnable* interface in an Espresso program, you have to use `import Runnable;`.

So far so good; now what should the EVM know about the interface *Runnable*? Since it is an interface, and does not have any code, the simplest approach is to compile `Runnable.java` with the Espresso compiler, and place the `Runnable.j` file in the `Lib` folder. That way your EVM can find it. Of course that will not be sufficient; there is still nothing that links a subclass of *Runnable* to a thread.

If we want to mimic the Java approach, clearly, the following code should be legal:

```
new Thread(new ClassThatImplementsRunnable()).start();
```

So, obviously *Thread* needs a constructor that accepts a reference to a *Runnable*. However, this is not enough. Recall the implementation of the *start()* method:

```
public final void start() {
  this.run();
}
```

This method needs to change a little; If a *Thread* object was created by passing an instance of a *Runnable* to it, then it must remember the reference of the object that was passed to it (to its constructor) because rather than run the *run()* method in the *Thread* class, the *run()* method in the *Runnable* class must be executed.

It obvious solution involved a field of type *Runnable* and a little rewriting of the *start()* method in the *Thread* class.

Once you have changed the *Thread.java* in the `Include` directory, do not forget to compile it with the Espresso compiler, and place the corresponding `.j` file in the `Lib` directory. Once you have done that the following program should work:

```
import System;
import Thread;
```

```
public class MyRunnable implements Runnable {
  public void run() {
    System.out.println("Hello from a thread that was implemented" +
                       " using the Runnable Interface");
  }
}

public class ThreadTest {
  public static void main() {
    new Thread(new MyRunnable()).start();
    System.out.println("Hello from Main");
  }
}
```

5.8 What to do

Question 20: Implement the multi-threading for the EVM and test it.

5.9 Hand In

Use the command `./evmu tar 5` to create a `.tar` file. Hand in this file without renaming it on `csgfms.cs.unlv.edu` before the deadline.

Appendix A

Instruction Semantics

A.1 binOp()

These are the operations you must implement in the *binOp()* function. You might wish to consult the actual Java Virtual Machine byte code specification (`http://cat.nyu.edu/~meyer/jvmref/`) for more detail.

Please note, in the following instruction specifications I have been a little sloppy with respect to notation: If the top two elements on the stack are of type *DoubleValue*, we should specify the `dadd` instruction in the following way:

dadd	
Syntax:	`dadd`
Description:	Removes two *DoubleValue* objects off the stack, adds their **double** values, and pushes a new *DoubleValue* element on to the stack.
Stack Before:	... D_1 D_2
Stack After:	... D
Comments:	D_1 and D_2 are of type *DoubleValue*, and D is of type *DoubleValue* and the actual **double** value of D is $D_1.getValue() + D_2.getValue()$.

but we will use the following short hand notation:

dadd	
Syntax:	`dadd`
Description:	Removes two **doubles** from the stack and pushes a new **double** onto the stack.
Stack Before:	... O_1 O_2
Stack After:	... O
Comments:	$O_1.type() = O_2.type() = Value.s\_double$ $O.type() = Value.s\_double$ $O.getValue() = O_1.getValue() + O_2.getValue()$

Note, of course, that you cannot make the assignment $O.getValue() = ...$, you have to create a new *DoubleValue* object and pass it the correct value to its constructor:

```
new DoubleValue(((DoubleValue)O2).getValue() + ((DoubleValue)O1).getValue());
```

Following are specifications of the rest of the instructions you must implement in *binOp()*.

A.1.1 Instructions on Double Values

dadd	
Syntax:	`dadd`
Description:	Removes two **doubles** from the stack and pushes a new **double** comprising their sum onto the stack.
Stack Before:	... O_1 O_2
Stack After:	... O
Comments:	$O.type() = O_1.type() = O_2.type() = Value.s\_double$ $O.getValue() = O_1.getValue() + O_2.getValue()$

dsub	
Syntax:	`dsub`
Description:	Removes two **doubles** from the stack and pushes a new **double** comprising their difference onto the stack.
Stack Before:	... O_1 O_2
Stack After:	... O
Comments:	$O.type() = O_1.type() = O_2.type() = Value.s\_double$ $O.getValue() = O_1.getValue() - O_2.getValue()$

ddiv	
Syntax:	`ddiv`
Description:	Removes two **doubles** from the stack and pushes a new **double**, obtained by dividing the first by the second number, onto the stack.
Stack Before:	... O_1 O_2
Stack After:	... O
Comments:	$O.type() = O_1.type() = O_2.type() = Value.s\_double$ $O.getValue() = O_1.getValue()/O_2.getValue()$

dmul	
Syntax:	`dmul`
Description:	Removes two **doubles** from the stack and pushes a new **double** comprising their product onto the stack.
Stack Before:	... O_1 O_2
Stack After:	... O
Comments:	$O.type() = O_1.type() = O_2.type() = Value.s\_double$ $O.getValue() = O_1.getValue() * O_2.getValue()$

drem	
Syntax:	`drem`
Description:	Removes two **doubles** from the stack and pushes a new **double** comprising the remainder of the division of the first number by the second onto the stack.
Stack Before:	... O_1 O_2
Stack After:	... O
Comments:	$O_1.type() = O_2.type() = Value.s\_double$ $O.getValue() = O_1.getValue()\%O_2.getValue()$

A.1.2 Instructions on Float Values

fadd	
Syntax:	`fadd`
Description:	Removes two **floats** from the stack and pushes a new **float** comprising their sum onto the stack.
Stack Before:	... O_1 O_2
Stack After:	... O
Comments:	$O.type() = O_1.type() = O_2.type() = Value.s\_float$ $O.getValue() = O_1.getValue() + O_2.getValue()$

fsub	
Syntax:	`fsub`
Description:	Removes two **floats** from the stack and pushes a new **float** comprising their difference onto the stack.
Stack Before:	... O_1 O_2
Stack After:	... O
Comments:	$O.type() = O_1.type() = O_2.type() = Value.s\_float$ $O.getValue() = O_1.getValue() - O_2.getValue()$

fdiv	
Syntax:	`fdiv`
Description:	Removes two **floats** from the stack and pushes a new **float**, with the value obtained by dividing the first by the second, onto the stack.
Stack Before:	... O_1 O_2
Stack After:	... O
Comments:	$O.type() = O_1.type() = O_2.type() = Value.s\_float$ $O.getValue() = O_1.getValue()/O_2.getValue()$

fmul	
Syntax:	`fmul`
Description:	Removes two **floats** from the stack and pushes a new **float** comprising their product onto the stack.
Stack Before:	... O_1 O_2
Stack After:	... O
Comments:	$O.type() = O_1.type() = O_2.type() = Value.s\_float$ $O.getValue() = O_1.getValue() * O_2.getValue()$

frem

Syntax:	`frem`
Description:	Removes two **floats** from the stack and pushes a new **float** comprising the remainder of dividing the first by the second onto the stack.
Stack Before:	... O_1 O_2
Stack After:	... O
Comments:	$O.type() = O_1.type() = O_2.type() = Value.s\_float$ $O.getValue() = O_1.getValue()\%O_2.getValue()$

A.1.3 Instructions on Long Values

ladd

Syntax:	`ladd`
Description:	Removes two **longs** from the stack and pushes a new **long** comprising their sum onto the stack.
Stack Before:	... O_1 O_2
Stack After:	... O
Comments:	$O.type() = O_1.type() = O_2.type() = Value.s\_long$ $O.getValue() = O_1.getValue() + O_2.getValue()$

lsub

Syntax:	`lsub`
Description:	Removes two **longs** from the stack and pushes a new **long** comprising their difference onto the stack.
Stack Before:	... O_1 O_2
Stack After:	... O
Comments:	$O.type() = O_1.type() = O_2.type() = Value.s\_long$ $O.getValue() = O_1.getValue() - O_2.getValue()$

ldiv

Syntax:	`ldiv`
Description:	Removes two **longs** from the stack and pushes a new **long** comprising the value obtained by dividing the first by the second onto the stack.
Stack Before:	... O_1 O_2
Stack After:	... O
Comments:	$O.type() = O_1.type() = O_2.type() = Value.s\_long$ $O.getValue() = O_1.getValue()/O_2.getValue()$

lmul

Syntax:	`lmul`
Description:	Removes two **longs** from the stack and pushes a new **long** comprising their product onto the stack.
Stack Before:	... O_1 O_2
Stack After:	... O
Comments:	$O.type() = O_1.type() = O_2.type() = Value.s\_long$ $O.getValue() = O_1.getValue() * O_2.getValue()$

lrem	
Syntax:	`lrem`
Description:	Removes two **longs** from the stack and pushes a new **long** comprising the remainder from the division of the first by the second onto the stack.
Stack Before:	... O_1 O_2
Stack After:	... O
Comments:	$O.type() = O_1.type() = O_2.type() = Value.s\_long$ $O.getValue() = O_1.getValue()\%O_2.getValue()$

A.1.4 Instructions on Integer Values

iadd	
Syntax:	`iadd`
Description:	Removes two **integers** from the stack and pushes a new **integer** comprising the sum onto the stack.
Stack Before:	... O_1 O_2
Stack After:	... O
Comments:	$O.type() = O_1.type() = O_2.type() = Value.s\_integer$ $O.getValue() = O_1.getValue() + O_2.getValue()$

isub	
Syntax:	`isub`
Description:	Removes two **integers** from the stack and pushes a new **integer** comprising their difference onto the stack.
Stack Before:	... O_1 O_2
Stack After:	... O
Comments:	$O.type() = O_1.type() = O_2.type() = Value.s\_integer$ $O.getValue() = O_1.getValue() - O_2.getValue()$

idiv	
Syntax:	`idiv`
Description:	Removes two **integers** from the stack and pushes a new **integer** with the value obtained by dividing the first by the second onto the stack.
Stack Before:	... O_1 O_2
Stack After:	... O
Comments:	$O.type() = O_1.type() = O_2.type() = Value.s\_integer$ $O.getValue() = O_1.getValue()/O_2.getValue()$

imul	
Syntax:	`imul`
Description:	Removes two **integers** from the stack and pushes a new **integer** comprising their product onto the stack.
Stack Before:	... O_1 O_2
Stack After:	... O
Comments:	$O.type() = O_1.type() = O_2.type() = Value.s\_integer$ $O.getValue() = O_1.getValue() * O_2.getValue()$

irem

Syntax:	`irem`
Description:	Removes two **integers** from the stack and pushes a new **integer** comprising the remainder of the division of the first by the second onto the stack.
Stack Before:	... O_1 O_2
Stack After:	... O
Comments:	$O.type() = O_1.type() = O_2.type() = Value.s\_integer$ $O.getValue() = O_1.getValue()\%O_2.getValue()$

A.2 swap() & pop()

There is only one instruction for swapping values, and it is `swap`. The semantics for this instruction is as follows:

swap	
Syntax:	`swap`
Description:	Swaps the top two stack values. Neither of these may be of type **long** or **double**
Stack Before:	... O_1 O_2
Stack After:	... O_2 O_1
Comments:	$O_1.type() \neq Value.s\_double \wedge O_1.type() \neq Value.s\_long$ $O_2.type() \neq Value.s\_double \wedge O_2.type() \neq Value.s\_long$

There are two different popping operations - they both do exactly the same:

pop	
Syntax:	`pop`
Description:	Removes the top element of the stack
Stack Before:	... O
Stack After:	...
Comments:	

pop2	
Syntax:	`pop2`
Description:	Removes the top element of the stack
Stack Before:	... O
Stack After:	...
Comments:	In the real JVm this pops two 32-bit words but since we work with Value objects, just remove one!

A.3 negate()

There are only 4 instructions for negating: `ineg`, `lneg`, `fneg`, and `dneg`.

ineg

Syntax:	`ineg`
Description:	Removes one **integer** from the stack and pushes the corresponding negated **integer** value onto the stack.
Stack Before:	$\ldots O_1$
Stack After:	$\ldots O$
Comments:	$O.type() = O_1.type() = Value.s\_integer$ $O.getValue() = -O_1.getValue()$

lneg

Syntax:	`lneg`
Description:	Removes one **long** from the stack and pushes the corresponding negated **long** value onto the stack.
Stack Before:	$\ldots O_1$
Stack After:	$\ldots O$
Comments:	$O.type() = O_1.type() = Value.s\_long$ $O.getValue() = -O_1.getValue()$

fneg

Syntax:	`fneg`
Description:	Removes one **float** from the stack and pushes the corresponding negated **float** value onto the stack.
Stack Before:	$\ldots O_1$
Stack After:	$\ldots O$
Comments:	$O.type() = O_1.type() = Value.s\_float$ $O.getValue() = -O_1.getValue()$

dneg

Syntax:	`dneg`
Description:	Removes one **double** from the stack and pushes the corresponding negated **double** value onto the stack.
Stack Before:	$\ldots O_1$
Stack After:	$\ldots O$
Comments:	$O.type() = O_1.type() = Value.s\_double$ $O.getValue() = -O_1.getValue()$

A.4 cmp()

fcmpl/fcmpg

Syntax:	`fcmpl/fcmpg`
Description:	Removes two **floats** of the stack and pushes -1, 0, or 1 onto the stack.
Stack Before:	... O_1 O_2
Stack After:	... O
Comments:	$O_1.type() = O_2.type() = Value.s\_float$ $O.type() = Value.s\_integer$ $O.getValue() = \begin{cases} 1 & : \quad O_1.getValue() > O2.getValue() \\ 0 & : \quad O_1.getValue() = O2.getValue() \\ -1 & : \quad O_1.getValue() < O2.getValue() \end{cases}$

dcmpl/dcmpg

Syntax:	`dcmpl/dcmpg`
Description:	Removes two **doubles** of the stack and pushes -1, 0, or 1 onto the stack.
Stack Before:	... O_1 O_2
Stack After:	... O
Comments:	$O_1.type() = O_2.type() = Value.s\_double$ $O.type() = Value.s\_integer$ $O.getValue() = \begin{cases} 1 & : \quad O_1.getValue() > O2.getValue() \\ 0 & : \quad O_1.getValue() = O2.getValue() \\ -1 & : \quad O_1.getValue() < O2.getValue() \end{cases}$

lcmp

Syntax:	`lcmp`
Description:	Removes two **longs** of the stack and pushes -1, 0, or 1 onto the stack.
Stack Before:	... O_1 O_2
Stack After:	... O
Comments:	$O_1.type() = O_2.type() = Value.s\_long$ $O.type() = Value.s\_integer$ $O.getValue() = \begin{cases} 1 & : \quad O_1.getValue() > O2.getValue() \\ 0 & : \quad O_1.getValue() = O2.getValue() \\ -1 & : \quad O_1.getValue() < O2.getValue() \end{cases}$

A.5 two()

i2d

Syntax:	`i2d`
Description:	Removes an **integer** from the stack, converts it to a **double** and pushes this **double** value on to the stack.
Stack Before:	$\ldots O_1$
Stack After:	$\ldots O$
Comments:	$O_1.type() = Value.s\_integer$ $O.type() = Value.s\_double$ $O.getValue() = (double)O_1.getValue()$

i2f

Syntax:	`i2f`
Description:	Removes an **integer** from the stack, converts it to a **float** and pushes this **float** value on to the stack.
Stack Before:	$\ldots O_1$
Stack After:	$\ldots O$
Comments:	$O_1.type() = Value.s\_integer$ $O.type() = Value.s\_float$ $O.getValue() = (float)O_1.getValue()$

i2l

Syntax:	`i2l`
Description:	Removes an **integer** from the stack, converts it to a **long** and pushes this **long** value on to the stack.
Stack Before:	$\ldots O_1$
Stack After:	$\ldots O$
Comments:	$O_1.type() = Value.s\_integer$ $O.type() = Value.s\_long$ $O.getValue() = (long)O_1.getValue()$

i2b

Syntax:	`i2b`
Description:	Removes an **integer** from the stack, zeros out the top 24 bits, resulting in an 8 bit **byte** value, sign extends it to an **integer** and pushes it onto the stack.
Stack Before:	$\ldots O_1$
Stack After:	$\ldots O$
Comments:	$O_1.type() = Value.s\_integer$ $O.type() = Value.s\_integer$ $O.getValue() = (integer)((byte)O_1.getValue())$

i2c

Syntax:	`i2c`
Description:	Removes an **integer** from the stack, zeros out the top 16 bits, resulting in an 16 bit **char** value, extends it to an **integer** and pushes it onto the stack.
Stack Before:	$\ldots O_1$
Stack After:	$\ldots O$
Comments:	$O_1.type() = Value.s\_integer$ $O.type() = Value.s\_integer$ $O.getValue() = (integer)((char)O_1.getValue())$

i2s

Syntax:	`i2s`
Description:	Removes an **integer** from the stack, zeros out the top 16 bits, resulting in an 16 bit **short** value, sign extends it to an **integer** and pushes it onto the stack.
Stack Before:	$\ldots O_1$
Stack After:	$\ldots O$
Comments:	$O_1.type() = Value.s\_integer$ $O.type() = Value.s\_integer$ $O.getValue() = (integer)((short)O_1.getValue())$

d2i

Syntax:	`d2i`
Description:	Removes a **double** from the stack, converts it to an **integer** and pushes this **integer** value on to the stack.
Stack Before:	$\ldots O_1$
Stack After:	$\ldots O$
Comments:	$O_1.type() = Value.s\_double$ $O.type() = Value.s\_integer$ $O.getValue() = (integer)O_1.getValue()$

d2f

Syntax:	`d2f`
Description:	Removes a **double** from the stack, converts it to a **float** and pushes this **float** value on to the stack.
Stack Before:	$\ldots O_1$
Stack After:	$\ldots O$
Comments:	$O_1.type() = Value.s\_double$ $O.type() = Value.s\_float$ $O.getValue() = (float)O_1.getValue()$

d2l

Syntax:	`d2l`
Description:	Removes a **double** from the stack, converts it to a **long** and pushes this **long** value on to the stack.
Stack Before:	$\ldots O_1$
Stack After:	$\ldots O$
Comments:	$O_1.type() = Value.s\_double$ $O.type() = Value.s\_long$ $O.getValue() = (long)O_1.getValue()$

f2i

Syntax:	`f2i`
Description:	Removes a **float** from the stack, converts it to an **integer** and pushes this **integer** value on to the stack.
Stack Before:	$\ldots O_1$
Stack After:	$\ldots O$
Comments:	$O_1.type() = Value.s\_float$ $O.type() = Value.s\_integer$ $O.getValue() = (integer)O_1.getValue()$

f2d

Syntax:	`f2d`
Description:	Removes a **float** from the stack, converts it to a **double** and pushes this **double** value on to the stack.
Stack Before:	$\ldots O_1$
Stack After:	$\ldots O$
Comments:	$O_1.type() = Value.s\_float$ $O.type() = Value.s\_double$ $O.getValue() = (double)O_1.getValue()$

f2l

Syntax:	`f2l`
Description:	Removes a **float** from the stack, converts it to a **long** and pushes this **long** value on to the stack.
Stack Before:	$\ldots O_1$
Stack After:	$\ldots O$
Comments:	$O_1.type() = Value.s\_float$ $O.type() = Value.s\_long$ $O.getValue() = (long)O_1.getValue()$

l2i

Syntax:	`l2i`
Description:	Removes a **long** from the stack, converts it to an **integer** and pushes this **integer** value on to the stack.
Stack Before:	$\ldots O_1$
Stack After:	$\ldots O$
Comments:	$O_1.type() = Value.s\_long$ $O.type() = Value.s\_integer$ $O.getValue() = (integer)O_1.getValue()$

l2d

Syntax:	`l2d`
Description:	Removes a **long** from the stack, converts it to a **double** and pushes this **double** value on to the stack.
Stack Before:	$\ldots O_1$
Stack After:	$\ldots O$
Comments:	$O_1.type() = Value.s\_long$ $O.type() = Value.s\_double$ $O.getValue() = (double)O_1.getValue()$

l2f

Syntax:	`l2f`
Description:	Removes a **long** from the stack, converts it to a **float** and pushes this **float** value on to the stack.
Stack Before:	$\ldots O_1$
Stack After:	$\ldots O$
Comments:	$O_1.type() = Value.s\_long$ $O.type() = Value.s\_float$ $O.getValue() = (float)O_1.getValue()$

A.6 logic()

iand

Syntax:	`iand`
Description:	Removes two **integers** from the stack and pushes the bitwise `and` of these values back on the stack as an **integer**.
Stack Before:	... O_1 O_2
Stack After:	... O
Comments:	$O.type() = O_1.type() = O_2.type() = Value.s\_integer$ $O.getValue() = O_1.getValue()$ & $O_2.getValue()$

ior

Syntax:	`ior`
Description:	Removes two **integers** from the stack and pushes the bitwise `or` of these values back on the stack as an **integer**.
Stack Before:	... O_1 O_2
Stack After:	... O
Comments:	$O.type() = O_1.type() = O_2.type() = Value.s\_integer$ $O.getValue() = O_1.getValue()$ | $O_2.getValue()$

ixor

Syntax:	`ixor`
Description:	Removes two **integers** from the stack and pushes the bitwise `xor` of these values back on the stack as an **integer**.
Stack Before:	... O_1 O_2
Stack After:	... O
Comments:	$O.type() = O_1.type() = O_2.type() = Value.s\_integer$ $O.getValue() = O_1.getValue()$ ^ $O_2.getValue()$

land

Syntax:	`land`
Description:	Removes two **longs** from the stack and pushes the bitwise `and` of these values back on the stack as a **long**.
Stack Before:	... O_1 O_2
Stack After:	... O
Comments:	$O.type() = O_1.type() = O_2.type() = Value.s\_long$ $O.getValue() = O_1.getValue()$ & $O_2.getValue()$

lor

Syntax:	`lor`
Description:	Removes two **longs** from the stack and pushes the bitwise `or` of these values back on the stack as an **long**.
Stack Before:	... O_1 O_2
Stack After:	... O
Comments:	$O.type() = O_1.type() = O_2.type() = Value.s\_long$ $O.getValue() = O_1.getValue()$ | $O_2.getValue()$

lxor

Syntax:	`lxor`
Description:	Removes two **longs** from the stack and pushes the bitwise `xor` of these values back on the stack as an **long**.
Stack Before:	... O_1 O_2
Stack After:	... O
Comments:	$O.type() = O_1.type() = O_2.type() = Value.s\_long$ $O.getValue() = O_1.getValue()$ ^ $O_2.getValue()$

A.7 shift()

ishl

Syntax:	`ishl`
Description:	Removes two **integers** from the stack and shifts the second value left by the amount indicated by the 5 low bits of the first value. This is the same as multiplying by 2^s where s is the value of the lower 5 bits.
Stack Before:	... O_1 O_2
Stack After:	... O
Comments:	$O_1.type() = O_2.type() = Value.s\_integer$ $O.type() = Values.s\_integer$ $O.getValue() = O_2.getValue() << (31 \ \& \ O_1.getValue())$

ishr

Syntax:	`ishr`
Description:	Removes two **integers** from the stack and shifts the second value right by the amount indicated by the 5 low bits of the first value. The sign is kept. This is the same as dividing by 2^s where s is the value of the lower 5 bits.
Stack Before:	... O_1 O_2
Stack After:	... O
Comments:	$O_1.type() = O_2.type() = Value.s\_integer$ $O.type() = Values.s\_integer$ $O.getValue() = O_2.getValue() >> (31 \ \& \ O_1.getValue())$

iushr

Syntax:	`iushr`
Description:	Removes two **integers** from the stack and shifts the second value right by the amount indicated by the 5 low bits of the first value. The sign is ignored.
Stack Before:	... O_1 O_2
Stack After:	... O
Comments:	$O_1.type() = O_2.type() = Value.s\_integer$ $O.type() = Values.s\_integer$ $O.getValue() = O_2.getValue() >>> (31 \ \& \ O_1.getValue())$

lshl

Syntax:	`lshl`
Description:	Removes an **integer** and a **long** from the stack and shifts the second value left by the amount indicated by the 6 low bits of the first value. This is the same as multiplying by 2^s where s is the value of the lower 6 bits of the first value.
Stack Before:	... O_1 O_2
Stack After:	... O
Comments:	$O_1.type() = Value.s\_integer$ $O_2.type() = Value.s\_long$ $O.type() = Values.s\_long$ $O.getValue() = O_2.getValue() << (63 \ \& \ O_1.getValue())$

lshr

Syntax:	`lshr`
Description:	Removes an **integer** and a **long** from the stack and shifts the second value right by the amount indicated by the 6 low bits of the first value. The sign is kept. This is the same as dividing by 2^s where s is the value of the lower 6 bits of the first value.
Stack Before:	... O_1 O_2
Stack After:	... O
Comments:	$O_1.type() = Value.s\_integer$ $O_2.type() = Value.s\_long$ $O.type() = Values.s\_long$ $O.getValue() = O_2.getValue() >> (63 \ \& \ O_1.getValue())$

lushr

Syntax:	`lushr`
Description:	Removes an **integer** and a **long** from the stack and shifts the second value right by the amount indicated by the 6 low bits of the first value. The sign is ignored.
Stack Before:	... O_1 O_2
Stack After:	... O
Comments:	$O_1.type() = Value.s\_integer$ $O_2.type() = Value.s\_long$ $O.type() = Values.s\_long$ $O.getValue() = O_2.getValue() >>> (63 \ \& \ O_1.getValue())$

A.8 Loading Constants to the Stack

Here is the semantics for the various instructions for loading constants onto the operand stack.

aconst_null

Syntax:	`aconst_null`
Description:	Loads a reference value representing the null value on to the stack.
Stack Before:	...
Stack After:	... O
Comments:	$O.type() Value. = s\_reference$ $O.getValue() = null$

dconst_0

Syntax:	`dconst_0`
Description:	Loads the constant 0.0 as a **double** onto the stack.
Stack Before:	...
Stack After:	... O
Comments:	$O.type = Value.s\_double$ $O.getValue() = 0.0$

dconst_1

Syntax:	`dconst_1`
Description:	Loads the constant 1.0 as a **double** onto the stack.
Stack Before:	...
Stack After:	... O
Comments:	$O.type = Value.s\_double$ $O.getValue() = 1.0$

fconst_0

Syntax:	`fconst_0`
Description:	Loads the constant 0.0 as a **float** onto the stack.
Stack Before:	...
Stack After:	... O
Comments:	$O.type = Value.s\_float$ $O.getValue() = 0.0$

fconst_1

Syntax:	`fconst_1`
Description:	Loads the constant 1.0 as a **float** onto the stack.
Stack Before:	...
Stack After:	... O
Comments:	$O.type = Value.s\_float$ $O.getValue() = 1.0$

fconst_2

Syntax:	`fconst_2`
Description:	Loads the constant 2.0 as a **float** onto the stack.
Stack Before:	...
Stack After:	... O
Comments:	$O.type = Value.s\_float$ $O.getValue() = 2.0$

lconst_0

Syntax:	`fconst_0`
Description:	Loads the constant 0 as a **long** onto the stack.
Stack Before:	...
Stack After:	... O
Comments:	$O.type = Value.s\_long$ $O.getValue() = 0$

lconst_1

Syntax:	`fconst_1`
Description:	Loads the constant 1 as a **long** onto the stack.
Stack Before:	...
Stack After:	... O
Comments:	$O.type = Value.s\_long$ $O.getValue() = 1$

iconst_m1

Syntax:	`iconst_m1`
Description:	Loads the constant -1 as a **integer** onto the stack.
Stack Before:	...
Stack After:	... O
Comments:	$O.type = Value.s\_integer$ $O.getValue() = -1$

iconst_0

Syntax:	`iconst_0`
Description:	Loads the constant 0 as a **integer** onto the stack.
Stack Before:	...
Stack After:	... O
Comments:	$O.type = Value.s\_integer$ $O.getValue() = 0$

iconst_1

Syntax:	`iconst_1`
Description:	Loads the constant 1 as a **integer** onto the stack.
Stack Before:	...
Stack After:	... O
Comments:	$O.type = Value.s\_integer$ $O.getValue() = 1$

iconst_2

Syntax:	`iconst_2`
Description:	Loads the constant 2 as a **integer** onto the stack.
Stack Before:	...
Stack After:	... O
Comments:	$O.type = Value.s\_integer$ $O.getValue() = 2$

iconst_3

Syntax:	`iconst_3`
Description:	Loads the constant 3 as a **integer** onto the stack.
Stack Before:	...
Stack After:	... O
Comments:	$O.type = Value.s\_integer$ $O.getValue() = 3$

iconst_4

Syntax:	`iconst_4`
Description:	Loads the constant 4 as a **integer** onto the stack.
Stack Before:	...
Stack After:	... O
Comments:	$O.type = Value.s\_integer$ $O.getValue() = 4$

iconst_5

Syntax:	`iconst_5`
Description:	Loads the constant 5 as a **integer** onto the stack.
Stack Before:	...
Stack After:	... O
Comments:	$O.type = Value.s\_integer$ $O.getValue() = 5$

ldc/ldc_w/ldc2_w

Syntax:	`ldc/ldc_w/ldc2_w val`
Description:	Pushes a constant value onto the stack. The type of the value depends on the instruction type.
Stack Before:	...
Stack After:	... O
Comments:	$O.type() \in \{Value.s\_integer, Value.s\_float, Value.s\_double, Value.s\_long, Value.s\_string\}$ $O.getValue() =$ `val`

bipush

Syntax:	`bipush val`
Description:	Pushes a constant **byte** value on to the stack as an **integer.**
Stack Before:	...
Stack After:	... O
Comments:	$O.type() = Value.s\_integer$ $O.getValue() =$ `val`

sipush

Syntax:	`sipush val`
Description:	Pushes a constant **short** value on to the stack as an **integer.**
Stack Before:	...
Stack After:	... O
Comments:	$O.type() = Value.s\_integer$ $O.getValue() =$ `val`

A.9 Loading and Storing Values

In the following *A[n]* refers to the *variables* array from the *Activation* class at index *n*.

aload

Syntax:	`aload n`
Description:	Loads the content of address n of the current activation onto the operand stack. The element must be a reference value.
Stack Before:	...
Stack After:	... O
Comments:	$O.type() = A[\texttt{n}].type() = Value.s\_reference$ $O.getValue() = A[\texttt{n}]$

astore

Syntax:	`astore n`
Description:	Stores the element on the top of the operand stack into address n of the current activation. The element must be a reference value.
Stack Before:	... O
Stack After:	...
Comments:	$O.type() = A[\texttt{n}].type() = Value.s\_reference$ $A[\texttt{n}] = O.getValue()$

dload

Syntax:	`dload n`
Description:	Loads the content of address n of the current activation onto the operand stack. The element must be a **double** value.
Stack Before:	...
Stack After:	... O
Comments:	$O.type() = A[\texttt{n}].type() = Value.s\_double$ $O.getValue() = A[\texttt{n}]$

dstore

Syntax:	`dstore n`
Description:	Stores the element on the top of the operand stack into address n of the current activation. The element must be a **double** value.
Stack Before:	... O
Stack After:	...
Comments:	$O.type() = A[\texttt{n}].type() = Value.s\_double$ $A[\texttt{n}] = O.getValue()$

fload

Syntax:	`fload n`
Description:	Loads the content of address n of the current activation onto the operand stack. The element must be a **float** value.
Stack Before:	...
Stack After:	... O
Comments:	$O.type() = A[\texttt{n}].type() = Value.s\_float$ $O.getValue() = A[\texttt{n}]$

fstore

Syntax:	`fstore n`
Description:	Stores the element on the top of the operand stack into address n of the current activation. The element must be a **float** value.
Stack Before:	$\ldots O$
Stack After:	$\ldots$
Comments:	$O.type() = A[\texttt{n}].type() = Value.s\_float$ $A[\texttt{n}] = O.getValue()$

iload

Syntax:	`iload n`
Description:	Loads the content of address n of the current activation onto the operand stack. The element must be an **integer** value.
Stack Before:	$\ldots$
Stack After:	$\ldots O$
Comments:	$O.type() = A[\texttt{n}].type() = Value.s\_integer$ $O.getValue() = A[\texttt{n}]$

istore

Syntax:	`istore n`
Description:	Stores the element on the top of the operand stack into address n of the current activation. The element must be an **integer** value.
Stack Before:	$\ldots O$
Stack After:	$\ldots$
Comments:	$O.type() = A[\texttt{n}].type() = Value.s\_integer$ $A[\texttt{n}] = O.getValue()$

lload

Syntax:	`lload n`
Description:	Loads the content of address n of the current activation onto the operand stack. The element must be a **long** value.
Stack Before:	$\ldots$
Stack After:	$\ldots O$
Comments:	$O.type() = A[\texttt{n}].type() = Value.s\_long$ $O.getValue() = A[\texttt{n}]$

lstore

Syntax:	`lstore n`
Description:	Stores the element on the top of the operand stack into address n of the current activation. The element must be a **long** value.
Stack Before:	$\ldots O$
Stack After:	$\ldots$
Comments:	$O.type() = A[\texttt{n}].type() = Value.s\_long$ $A[\texttt{n}] = O.getValue()$

Xload_Y

Syntax: `Xload_Y`

Description: Loads the content of address n of the current activation onto the operand stack. The element must be of type **integer** if `X` is `i`, **long** if `X` is `l`, **float** if `X` is `f`, **double** if `X` is `d`, or a reference if `X` is `a`. `Y` must be one of `0`, `1`, `2`, or `3`.

Stack Before: $\ldots$

Stack After: $\ldots\ O$

Comments:

$$type = \begin{cases} Value.s\_integer & : \texttt{X} = \texttt{i} \\ Value.s\_float & : \texttt{X} = \texttt{f} \\ Value.s\_double & : \texttt{X} = \texttt{d} \\ Value.s\_long & : \texttt{X} = \texttt{l} \\ Value.s\_reference & : \texttt{X} = \texttt{a} \end{cases}$$

$\texttt{Y} \in \{0, 1, 2, 3\}$

$O.type() = A[\texttt{Y}].type() = type$

$O.getValue() = A[\texttt{Y}]$

Xstore_Y

Syntax: `Xstore_Y`

Description: Stores the element at the top of the operand stack at address n of the current activation. The element must be of type **integer** if `X` is `i`, **long** if `X` is `l`, **float** if `X` is `f`, **double** if `X` is `d`, or a reference if `X` is `a`. `Y` must be one of `0`, `1`, `2`, or `3`.

Stack Before: $\ldots\ O$

Stack After: $\ldots$

Comments:

$$type = \begin{cases} Value.s\_integer & : \texttt{X} = \texttt{i} \\ Value.s\_float & : \texttt{X} = \texttt{f} \\ Value.s\_double & : \texttt{X} = \texttt{d} \\ Value.s\_long & : \texttt{X} = \texttt{l} \\ Value.s\_reference & : \texttt{X} = \texttt{a} \end{cases}$$

$\texttt{Y} \in \{0, 1, 2, 3\}$

$O.type() = A[\texttt{Y}].type() = type$

$A[\texttt{Y}] = O.getValue()$

iinc

Syntax: `iinc n i`

Description: Increments $A[\texttt{n}]$ by `i`. $A[\texttt{n}]$ must be of **integer** type.

Stack Before: $\ldots$

Stack After: $\ldots$

Comments: $A[\texttt{n}].type() = Value.s\_integer$

$A[\texttt{n}] = A[\texttt{n}] + \texttt{i}$

A.10 Branching

In the following the term 'jump to label L' means 'set *pc* to the address of label L'. This address can be looked up using the *getLabelAddress()* on the current method (*currentMethod*). `@L` means the address of the label `L`.

ifeq

Syntax:	`ifeq L`
Description:	Jumps to label `L` if the top of the stack is the **integer** value 0.
Stack Before:	$\ldots O$
Stack After:	$\ldots$
Comments:	$O.type() = Value.s\_integer$ $pc = \begin{cases} \texttt{@L} & : \ O.getValue() = 0 \\ pc+1 & : \ \text{otherwise} \end{cases}$

ifne

Syntax:	`ifne L`
Description:	Jumps to label `L` if the top of the stack is not the **integer** value 0.
Stack Before:	$\ldots O$
Stack After:	$\ldots$
Comments:	$O.type() = Value.s\_integer$ $pc = \begin{cases} \texttt{@L} & : \ O.getValue() \neq 0 \\ pc+1 & : \ \text{otherwise} \end{cases}$

iflt

Syntax:	`iflt L`
Description:	Jumps to label `L` if the top of the stack is smaller than the **integer** value 0.
Stack Before:	$\ldots O$
Stack After:	$\ldots$
Comments:	$O.type() = Value.s\_integer$ $pc = \begin{cases} \texttt{@L} & : \ O.getValue() < 0 \\ pc+1 & : \ \text{otherwise} \end{cases}$

ifle

Syntax:	`ifle L`
Description:	Jumps to label `L` if the top of the stack is smaller than or equal to the **integer** value 0.
Stack Before:	$\ldots O$
Stack After:	$\ldots$
Comments:	$O.type() = Value.s\_integer$ $pc = \begin{cases} \texttt{@L} & : \ O.getValue() \leq 0 \\ pc+1 & : \ \text{otherwise} \end{cases}$

ifgt

Syntax: `ifgt L`

Description: Jumps to label `L` if the top of the stack is greater than the **integer** value 0.

Stack Before: ... O

Stack After: ...

Comments: $O.type() = Value.s\_integer$

$$pc = \begin{cases} \texttt{@L} & : \quad O.getValue() > 0 \\ pc + 1 & : \quad \text{otherwise} \end{cases}$$

ifge

Syntax: `ifge L`

Description: Jumps to label `L` if the top of the stack is greater than or equal to the **integer** value 0.

Stack Before: ... O

Stack After: ...

Comments: $O.type() = Value.s\_integer$

$$pc = \begin{cases} \texttt{@L} & : \quad O.getValue() \geq 0 \\ pc + 1 & : \quad \text{otherwise} \end{cases}$$

if_icmpeq

Syntax: `if_icmpeq L`

Description: Jumps to label `L` if the two **integer** values on top of the stack represent the same value.

Stack Before: ... O_1 O_2

Stack After: ...

Comments: $O_1.type() = O_2.type() = Value.s\_integer$

$$pc = \begin{cases} \texttt{@L} & : \quad O_1.getValue() = O_2.getValue() \\ pc + 1 & : \quad \text{otherwise} \end{cases}$$

if_icmpne

Syntax: `if_icmpne L`

Description: Jumps to label `L` if the two **integer** values on top of the stack do not represent the same value.

Stack Before: ... O_1 O_2

Stack After: ...

Comments: $O_1.type() = O_2.type() = Value.s\_integer$

$$pc = \begin{cases} \texttt{@L} & : \quad O_1.getValue() \neq O_2.getValue() \\ pc + 1 & : \quad \text{otherwise} \end{cases}$$

if_icmplt

Syntax: `if_icmplt L`

Description: Jumps to label `L` if the top **integer** value on the stack if greater than the one below it.

Stack Before: ... O_1 O_2

Stack After: ...

Comments: $O_1.type() = O_2.type() = Value.s\_integer$

$$pc = \begin{cases} \texttt{@L} & : \quad O_1.getValue() < O_2.getValue() \\ pc + 1 & : \quad \text{otherwise} \end{cases}$$

if_icmple

Syntax:	`if_icmple L`
Description:	Jumps to label `L` if the top **integer** value on the stack if greater than or equal to the one below it.
Stack Before:	... O_1 O_2
Stack After:	...
Comments:	$O_1.type() = O_2.type() = Value.s\_integer$ $pc = \begin{cases} \texttt{@L} & : \ O_1.getValue() <= O_2.getValue() \\ pc+1 & : \ \text{otherwise} \end{cases}$

if_icmpgt

Syntax:	`if_icmpgt L`
Description:	Jumps to label `L` if the top **integer** value on the stack if smaller than the one below it.
Stack Before:	... O_1 O_2
Stack After:	...
Comments:	$O_1.type() = O_2.type() = Value.s\_integer$ $pc = \begin{cases} \texttt{@L} & : \ O_1.getValue() > O_2.getValue() \\ pc+1 & : \ \text{otherwise} \end{cases}$

if_icmpge

Syntax:	`if_icmpge L`
Description:	Jumps to label `L` if the top **integer** value on the stack if smaller than or equal to the one below it.
Stack Before:	... O_1 O_2
Stack After:	...
Comments:	$O_1.type() = O_2.type() = Value.s\_integer$ $pc = \begin{cases} \texttt{@L} & : \ O_1.getValue() >= O_2.getValue() \\ pc+1 & : \ \text{otherwise} \end{cases}$

if_acmpeq

Syntax:	`if_acmpeq L`
Description:	Jumps to label `L` if the two reference values on top of the stack represent the same object.
Stack Before:	... O_1 O_2
Stack After:	...
Comments:	$O_1.type() = O_2.type() = Value.s\_reference$ $pc = \begin{cases} \texttt{@L} & : \ O_1.getValue() = O_2.getValue() \\ pc+1 & : \ \text{otherwise} \end{cases}$

if_acmpne

Syntax:	`if_acmpne L`
Description:	Jumps to label `L` if the two reference values on top of the stack do not represent the same object.
Stack Before:	... O_1 O_2
Stack After:	...
Comments:	$O_1.type() = O_2.type() = Value.s\_reference$ $pc = \begin{cases} @L & : \ O_1.getValue() \neq O_2.getValue() \\ pc+1 & : \ \text{otherwise} \end{cases}$

ifnull

Syntax:	`ifnull L`
Description:	Jumps to label `L` if the reference value on top of the stack represents the null value.
Stack Before:	... O_1
Stack After:	...
Comments:	$O_1.type() = Value.s\_reference$ $pc = \begin{cases} @L & : \ O_1.getValue() = null \\ pc+1 & : \ \text{otherwise} \end{cases}$

ifnonnull

Syntax:	`ifnonnull L`
Description:	Jumps to label `L` if the reference value on top of the stack does not represents the null value.
Stack Before:	... O_1
Stack After:	...
Comments:	$O_1.type() = Value.s\_reference$ $pc = \begin{cases} @L & : \ O_1.getValue() \neq null \\ pc+1 & : \ \text{otherwise} \end{cases}$

goto/goto_w

Syntax:	`goto/goto_w`
Description:	Jumps to label `L`.
Stack Before:	...
Stack After:	...
Comments:	$pc = @L$

A.11 Creating Objects

new

Syntax:	`new` *class-name*
Description:	Instantiates an object based on the class of name *class-name* and leaves a reference to this object on top of the stack.
Stack Before:	...
Stack After:	... O
Comments:	$O.type() = Value.s\_reference$

instanceof

Syntax:	`instanceof class`
Description:	Checks if the the reference value on the top of the stack represents an object that is an instance of `class`.
Stack Before:	... O
Stack After:	... V
Comments:	$O.type() = Value.s\_reference$ $V = \begin{cases} 0 & : \quad O = null \\ 1 & : \quad O \text{ is an instance of } \texttt{class} \\ 0 & : \quad otherwise \end{cases}$

checkcast

Syntax:	`checkcast class`
Description:	Checks if the reference value on the top of the stack represents an object that can legally be cast to `class`. If not, execution is terminated.
Stack Before:	... O
Stack After:	... O
Comments:	$O.type() = Value.s\_reference$ O can be cast to an object of type `class`

A.12 Manipulating Fields

In the following $O.class()$ refers to the class that O is an instance of. $O.staticFields()$ is the set of static fields on O, and $O.nonStaticFields()$ is the set of non-static fields of O, and $O.getStaticField(fieldName)$ refers to the static field named *fieldName*.
$O.getNonStaticField(fieldName)$ is the same for non-static fields.

putfield

Syntax:	`putfield class-name/field-name signature`
Description:	Assigns the value V to the field `field-name` in an object of type `class-name`. The stack should hold a reference to such an object as well as the value to be assigned. The type of the field should match that of the value on the stack. The field must not be static.
Stack Before:	$\ldots O\ V$
Stack After:	$\ldots$
Comments:	$O.type() = Value.s\_reference$ $O.class() = \texttt{class-name}$ $\texttt{field-name} \in O.nonStaticField()$ $O.getNonStaticField(\texttt{field-name}).type() = V.type() = signature.type()$ $O.getNonStaticField(\texttt{field-name}) = V$

getfield

Syntax:	`getfield class-name/field-name signature`
Description:	Copies the value of a field named `field-name` from object O onto the stack. O must be a reference to an object of type `class-name`, and the field must not be static.
Stack Before:	$\ldots O$
Stack After:	$\ldots V$
Comments:	$O.type() = Value.s\_reference$ $O.class() = \texttt{class-name}$ $\texttt{field-name} \in O.nonStaticField()$ $O.getNonStaticField(\texttt{field-name}).type() = V.type() = signature.type()$ $V = O.getNonStaticField(\texttt{field-name})$

putstatic

Syntax:	`putstatic class-name/field-name signature`
Description:	Assigns the value V to the static field `field-name` in class `class-name`. The top of the stack should be the value to be assigned. The type of the field should match that of the value on the stack. The field must be static.
Stack Before:	$\ldots V$
Stack After:	$\ldots$
Comments:	$\texttt{field-name} \in \texttt{class-name}.staticField()$ $\texttt{class-name}.getStaticField(\texttt{field-name}).type() = V.type()$ $= signature.type()$ $\texttt{class-name}.getStaticField(\texttt{field-name}) = V$

getstatic

Syntax:	`getstatic class-name/field-name signature`
Description:	Copies the value of a static field named `field-name` in class `class-name` onto the stack. The field must be static.
Stack Before:	...
Stack After:	... V
Comments:	$\texttt{field} - \texttt{name} \in O.staticField()$ $\texttt{class} - \texttt{name}.getStaticField(\texttt{field} - \texttt{name}).type() = V.type()$ $= signature.type()$ $V = \texttt{class} - \texttt{name}.getStaticField(\texttt{field} - \texttt{name})$

A.13 Invoking Methods and Constructors

In the following A is the *variables* array in the activation created with each invocation.

invokevirtual

Syntax:	`invokevirtual class-name/method-name(signature)return-type`
Description:	Invokes method `method-name` on object O. O must be of type `class-name` and take n parameters where the value of P_i can be assigned to the i'th formal parameter. A new activation is created and execution starts at address 0 of the method.
Stack Before:	$\ldots\ O\ P_1\ \ldots P_n$
Stack After:	$\ldots$
Comments:	$O.class() = \texttt{class} - \texttt{name}$ $signature[i].type() = P_i.type()$ $A[i] = P_i, i = 1, \ldots, n + 1$ $A[0] = this$

invokenonvirtual

Syntax:	`invokenonvirtual class-name/method-name(signature)return-type`
Description:	Invokes method `method-name` on object O. O must be of type `class-name` and take n parameters where the value of P_i can be assigned to the i'th formal parameter. A new activation is created and execution starts at address 0 of the method.
Stack Before:	$\ldots\ O\ P_1\ \ldots P_n$
Stack After:	$\ldots$
Comments:	$O.class() = \texttt{class} - \texttt{name}$ $signature[i].type() = P_i.type()$ $A[i] = P_i, i = 1, \ldots, n + 1$ $A[0] = this$

invokestatic

Syntax:	`invokestatic class-name/method-name(signature)return-type`
Description:	Invokes method `method-name` on class `class-name`. The method must take n parameters where the value of P_i can be assigned to the i'th formal parameter. A new activation is created and execution starts at address 0 of the method. The method must be static.
Stack Before:	$\ldots\ P_1\ \ldots P_n$
Stack After:	$\ldots$
Comments:	$A[i] = P_i, i = 0, \ldots, n$ $signature[i].type() = P_i.type()$

invokeinterface

Syntax:	`invokeinterface class-name/method-name(signature) n`
Description:	Invokes method `method-name` on object O. O must be of reference type, and represent a class that implements or inherits from someone who implements `class-name`. It must take `n` parameters where the value of P_i can be assigned to the i'th formal parameter. A new activation is created and execution starts at address 0 of the method.
Stack Before:	$\ldots\ O\ P_1 \ldots P_n$
Stack After:	$\ldots$
Comments:	$O.class()$ "extends or implements" $\texttt{class} - \texttt{name}$ $signature[i].type() = P_i.type()$ $A[i] = P_i, i = 1, \ldots, \texttt{n} + 1$ $A[0] = this$

A.14 Return

areturn	
Syntax:	`areturn`
Description:	Returns from a method or constructor call. The top element on the stack should be a reference to an object of the type the method returns. If this is a return from *main()* then the execution loop terminates. If it is a return from *<clinit>* the activation stack is popped and the execution loop terminates. Otherwise, *pc* is set to the return address, *currentMethod*, *activation* and *this_* are updated, and the activation stack popped.
Stack Before:	$\ldots O$
Stack After:	$\ldots O$
Comments:	$O.type() = Value.s\_reference$

ireturn	
Syntax:	`ireturn`
Description:	Returns from a method or constructor call. The top element on the stack should be an **integer** value. If this is a return from *main()* then the execution loop terminates. If it is a return from *<clinit>* the activation stack is popped and the execution loop terminates. Otherwise, *pc* is set to the return address, *currentMethod*, *activation* and *this_* are updated, and the activation stack popped.
Stack Before:	$\ldots O$
Stack After:	$\ldots O$
Comments:	$O.type() = Value.s\_integer$

freturn	
Syntax:	`freturn`
Description:	Returns from a method or constructor call. The top element on the stack should be a **float** value. If this is a return from *main()* then the execution loop terminates. If it is a return from *<clinit>* the activation stack is popped and the execution loop terminates. Otherwise, *pc* is set to the return address, *currentMethod*, *activation* and *this_* are updated, and the activation stack popped.
Stack Before:	$\ldots O$
Stack After:	$\ldots O$
Comments:	$O.type() = Value.s\_float$

lreturn	
Syntax:	`lreturn`
Description:	Returns from a method or constructor call. The top element on the stack should be a **long** value. If this is a return from *main()* then the execution loop terminates. If it is a return from *<clinit>* the activation stack is popped and the execution loop terminates. Otherwise, *pc* is set to the return address, *currentMethod*, *activation* and *this_* are updated, and the activation stack popped.
Stack Before:	$\ldots O$
Stack After:	$\ldots O$
Comments:	$O.type() = Value.s\_long$

dreturn

Syntax:	`dreturn`
Description:	Returns from a method or constructor call. The top element on the stack should be a **double** value. If this is a return from *main()* then the execution loop terminates. If it is a return from <*clinit*> the activation stack is popped and the execution loop terminates. Otherwise, *pc* is set to the return address, *currentMethod*, *activation* and *this_* are updated, and the activation stack popped.
Stack Before:	$\ldots O$
Stack After:	$\ldots O$
Comments:	$O.type() = Value.s\_double$

return

Syntax:	`return`
Description:	Returns from a method or constructor call. This instruction does not require a value on the stack as it is used for *void* methods. If this is a return from *main()* then the execution loop terminates. If it is a return from <*clinit*> the activation stack is popped and the execution loop terminates. Otherwise, *pc* is set to the return address, *currentMethod*, *activation* and *this_* are updated, and the activation stack popped.
Stack Before:	$\ldots$
Stack After:	$\ldots$
Comments:	