A very short introduction to JPF

Franco Raimondi

Department of Computer Science
School of Science and Technology
Middlesex University
http://www.rmnd.net

A very short introduction to JPF

Writing Java code

What happens when you write Java code?

```
public class Simple {

  static int plus (int a) {
    int b = 1;
    return a+b;
  }

  public static void main (String[] args) {
    System.out.println(plus(3));
  }
}
```

- Compile with `javac Simple.java`
- Run with `java Simple`

Java .class files

After compiling, you obtain a .class file. You can check the content with `javap -c -s -verbose Simple`:

```
[...]
0: iconst_1
1: istore_1
2: iload_0
3: iload_1
4: iadd
5: ireturn
[...]
```

Executing .class files

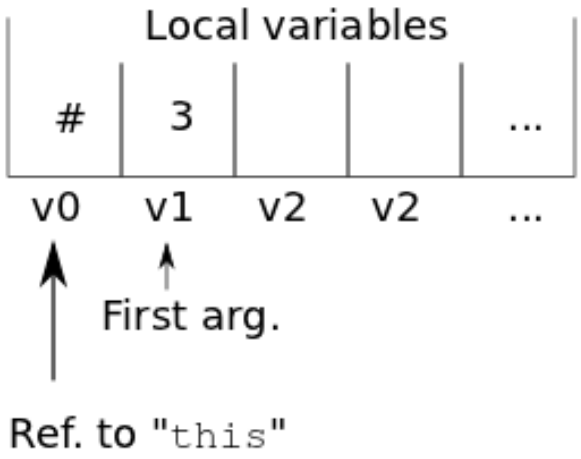
What happens when you run a .class file? The execution model of .class files is stack-based:

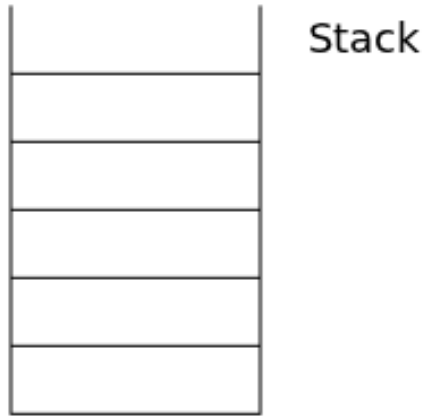
- Each method has an array of local variables and a "local" stack: this is called a *frame*.
- Each *thread* has a stack of frames.
- Each class contains a *constant pool*

Java bytecode example

```
int plus(int a)
  {
    int b = 1;
    return a+b;
  }
```

```
0:  iconst_1   // load constant 1 into stack
1:  istore_2   // store top stack in var 2
2:  iload_1    // load from var 1 to stack
3:  iload_2    // load from var 2 to stack
4:  iadd       // add 2 values on top of stack
5:  ireturn
```

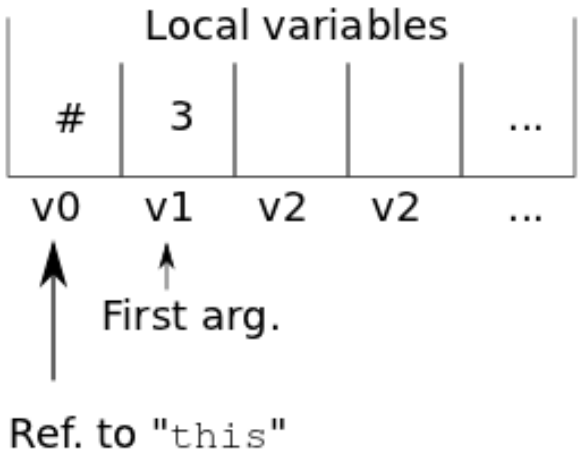


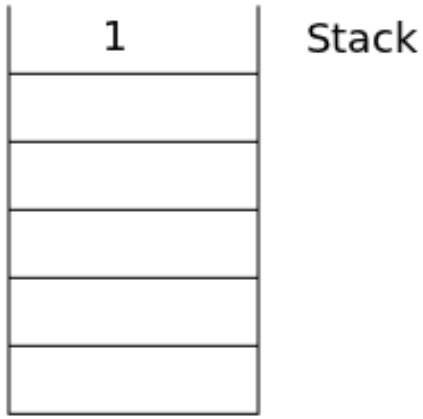


Java bytecode example - 2

```
int plus(int a)
  {
    int b = 1;
    return a+b;
  }
```

```
0:  iconst_1   // load constant 1 into stack
1:  istore_2   // store top stack in var 2
2:  iload_1    // load from var 1 to stack
3:  iload_2    // load from var 2 to stack
4:  iadd       // add 2 values on top of stack
5:  ireturn
```

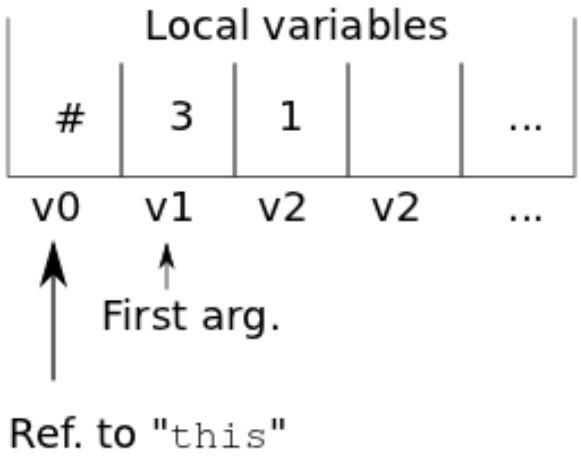


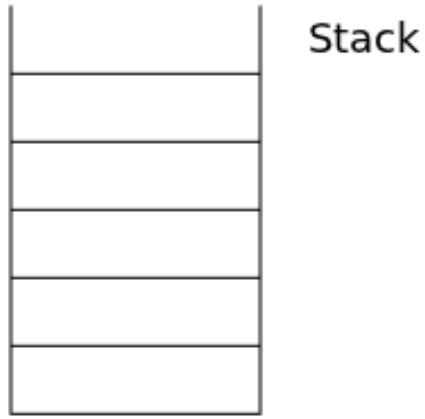


Java bytecode example - 3

```
int plus(int a)
  {
    int b = 1;
    return a+b;
  }
```

```
0:   iconst_1   // load constant 1 into stack
1:   istore_2   // store top stack in var 2
2:   iload_1    // load from var 1 to stack
3:   iload_2    // load from var 2 to stack
4:   iadd       // add 2 values on top of stack
5:   ireturn
```

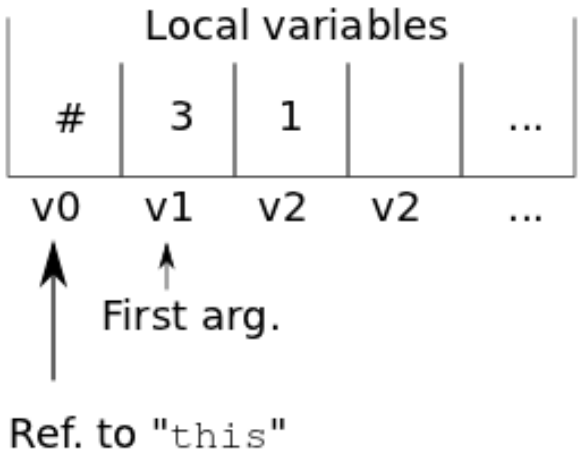


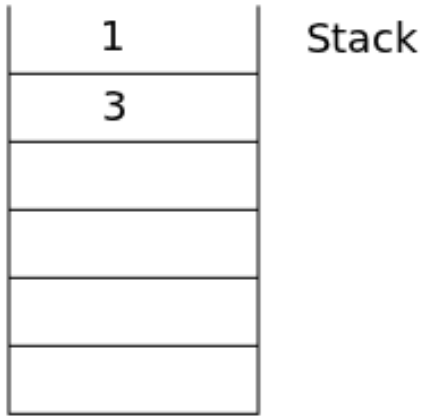


Java bytecode example - 4

```
int plus(int a)
  {
    int b = 1;
    return a+b;
  }
```

```
0:  iconst_1   // load constant 1 into stack
1:  istore_2   // store top stack in var 2
2:  iload_1    // load from var 1 to stack
3:  iload_2    // load from var 2 to stack
4:  iadd       // add 2 values on top of stack
5:  ireturn
```

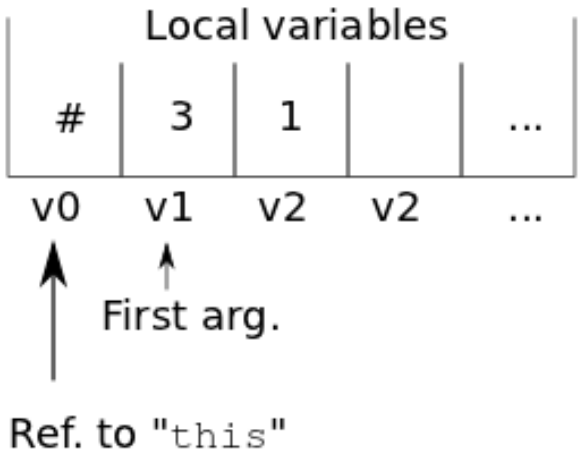


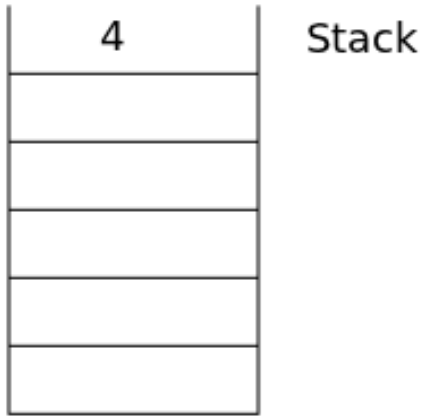


Java bytecode example - 5

```
int plus(int a)
  {
    int b = 1;
    return a+b;
  }
```

```
0:   iconst_1   // load constant 1 into stack
1:   istore_2   // store top stack in var 2
2:   iload_1    // load from var 1 to stack
3:   iload_2    // load from var 2 to stack
4:   iadd       // add 2 values on top of stack
5:   ireturn
```





Java Pathfinder

- JPF is a popular "model checker" for Java code. In its default configuration JPF detects unhandled exceptions, deadlocks, and races.
- JPF is essentially a customizable JVM. It reads .class files and replaces the default JVM.
- JPF is written in Java... so there is JVM running JPF, which is a JVM in itself.

`http://jpf.byu.edu/`

Choice generators and JPF states

- JPF creates a choice whenever multiple execution paths can arise (non-deterministic choices, user input, thread scheduling).
- **The byte-code comprised between two choices defines a JPF state.**
- JPF can store and explore states using various search strategies.

JPF: install and compile

(you need a JVM, I'm using 1.8, and ant)

```
$ hg clone  https://jpf.byu.edu/hg/jpf-core
$ cd jpf-core
$ ant
[... after a few seconds ...]
BUILD SUCCESSFUL
Total time: 11 seconds
```

Initial configuration

In your home directory, create a directory `~/.jpf`. In this directory, create a site.properties file similar to the following (change as appropriate):

```
# JPF site configuration
jpf-core = ${user.home}/path/to/jpf-core
extensions=${jpf-core}
```

Running JPF

- You need a .class file. There are some examples in the JPF distribution, we will use these.
- You need a configuration file to tell JPF what to do. Usually, if you want to verify the file `SomeClass.java`, you create a file called `SomeClass.jpf`

Example in `src/examples/Rand.jpf`:

```
target = Rand
cg.enumerate_random = true
report.console.property_violation=error,trace
```

Example Java class

```
public class Rand {
  public static void main (String[] args) {
    System.out.println("computing c = a/(b+a - 2)..");
    Random random = new Random();
    int a = random.nextInt(2);
    System.out.printf("a=%d\n", a);
    //... lots of code here
    int b = random.nextInt(3);
    System.out.printf("  b=%d       ,a=%d\n", b, a);
    int c = a/(b+a -2);
    System.out.printf("=>  c=%d     , b=%d, a=%d\n", c, b, a);
  }
}
```

Running JPF

From the directory `bin/`, run

`./jpf ../src/examples/Rand.jpf`

Check the output: various states are explored, and an error reported in one case:

```
computing c = a/(b+a - 2)..
a=0
  b=0        ,a=0
=>  c=0      , b=0, a=0
  b=1        ,a=0
=>  c=0      , b=1, a=0
  b=2        ,a=0

====================================================== error 1
gov.nasa.jpf.vm.NoUncaughtExceptionsProperty
java.lang.ArithmeticException: division by zero
at Rand.main(Rand.java:34)
```

Additional JPF features

- It is possible to write custom choice generators.
- It is possible to add *listeners*: for new states, but also for specific bytecode instructions.
- It is possible to write custom state matching mechanisms.
- It is possible to write custom search strategies (e.g.: DDFS for LTL verification).

Very simple listener

`src/main/gov/nasa/jpf/listener/SimpleDot.java` is an example of listener. To use it, add the following line to a .jpf configuration file: `listener=.listener.SimpleDot`
Run again jpf. Check the directory: you will get a Rand.dot file that you can plot.
[Additional details on the board]