1. State whether each of the following statements is true or false:

Circle your choice:

1) True/False The condition "Verify.*getBoolean*() == Verify.*getBoolean*()" always evaluates to "true" on the JPF Java Virtual Machine

2) True/False The following code snippet allows using JPF to find all solutions of the Boolean formula "x implies y":

```
boolean x = Verify.getBoolean();
boolean y = Verify.getBoolean();
if (!x || y) {
       System.out.println("x: " + x + "; y: " + y);
}
```

2. Consider implementing an *integer constraint solver* using the Java PathFinder model checker. Given a constraint on integer variables and domains that define the possible values that the variables can take, an *integer constraint solver* finds solution to the given constraint, i.e., the solver finds values for each variable such that the constraint evaluates to true for those values.

To illustrate, consider the constraint "$x + y == u + v$" and domains $x, y, u, v \in \{ 0, 1, 2 \}$. A solution to this constraint is "$x = 0; y = 0; u = 0; v = 0$".

Consider the following class that implements a candidate solution using a map from variable names to integer values:

```
public class CandidateSolution {
	Map<String, Integer> candidate = new HashMap<String, Integer>();

	void set(String var, int value) {
		// postcondition: update the map to set "var = value"
		candidate.put(var, value);
	}

	int value(String var) {
		// postcondition: returns the value of var in the map
		if (candidate.get(var) == null) {
			throw new RuntimeException("unknown variable: " + var);
		}
		return candidate.get(var);
	}

	public String toString() {
		return candidate.toString();
	}
}
```

To illustrate, the solution "$x = 0; y = 0; u = 0; v = 0$" is represented using the map "`{u=0, y=0, x=0, v=0}`".

Consider the following classes that define the basic abstract data types to implement an integer constraint solver:

```
public abstract class IntegerConstraint {
	// postcondition: evaluates this constraint on the given candidate
	// and returns true if and only if the candidate is a valid solution
	abstract boolean evaluate(CandidateSolution candidate);
}

public abstract class IntegerExpression {
	// postcondition: evaluates this expression on the given candidate
	// and returns the integer value of the expression
	abstract int evaluate(CandidateSolution candidate);
}
```

(a) Implement the method `evaluate` in the following class `EqualityConstraint`:

```
public class EqualityConstraint extends IntegerConstraint {
	IntegerExpression lhs, rhs;

	EqualityConstraint(IntegerExpression lhs, IntegerExpression rhs) {
		this.lhs = lhs;
		this.rhs = rhs;
	}

	boolean evaluate(CandidateSolution candidate) {
		// your code goes here









	}
}
```

(b) Implement the method `evaluate` in the following class `AdditionExpression`:

```
public class AdditionExpression extends IntegerExpression {
	String var1, var2;

	AdditionExpression(String v1, String v2) {
		var1 = v1;
		var2 = v2;
	}

	int evaluate(CandidateSolution candidate) {
		// your code goes here









	}
}
```

(c) Implement the method `solve` in the following class `IntegerConstraintSolver` to implement an integer constraint solver, where `ic` is the constraint to solve and `vars` is the set of variables that appear in the constraint:

```
import gov.nasa.jpf.jvm.Verify;

import java.util.HashSet;
import java.util.Set;

public class IntegerConstraintSolver {
	static void solve(IntegerConstraint ic, Set<String> vars) {
		// make non-deterministic initialization of variables in
		// vars to generate a candidate solution
		// assume: each variable can take one of the values 0, 1, 2









		// check whether the constraint is satisfied by the candidate;
		// print solution found




	}
```

(d) Implement the following method main in IntegerConstraintSolver to enumerate the solutions for the constraint "$x + y == u + v$" for domains $x, y, u, v \in \{ 0, 1, 2 \}$:

```
	public static void main(String[] a) {
		AdditionExpression lhs = new AdditionExpression("x", "y");
```

```
	}
}
```

3. Consider an array-based implementation of a *max-heap*.

(a) Complete the implementation of the following `repOk` method to include a check that the heap consists of unique values:

```
import java.util.HashSet;
import java.util.Set;

import gov.nasa.jpf.jvm.Verify;

public class HeapArray {
	int size; // number of elements in the heap
	Integer[] array; // heap elements

	boolean repOk() {
		// checks that array is non-null
		if (array == null) return false;

		// checks that size is within array bounds
		if (size < 0 || size > array.length) return false;

		for (int i = 0; i < size; i++) {
			// checks that elements are non-null
			if (array[i] == null) return false;
			// checks that array is heapified
			if (i > 0 &&
					array[i] > array[(i-1)/2])
				return false;
		}

		// checks that non-heap elements are null
		for (int i = size; i < array.length; i++) {
			if (array[i] != null) return false;
		}

		// no repetition of elements
		// your code goes here









	}
```

(b) Implement the following method `generate` in `HeapArray` to enable representation-level generation of heap-arrays using the Java PathFinder model checker for the bounds specified in the postcondition:

```
	static void generate() {
		// postcondition: enumerates heap-arrays using the following bounds:
		//	0 <= size <= 3
		//	each array element takes a value in { 0, 1, 2 }
		//	0 <= array.length <= 3

		// allocate objects and set fields non-deterministically using JPF









		// use repOk as a filter to check if the initialization generated
		// a valid heap array object; print valid objects on console









	}
```

You may assume the existence of the following `toString` method to pretty-print heap-arrays:

```
	public String toString() {
		StringBuffer sb = new StringBuffer();
		sb.append("[ ");
		for (Integer x: array) {
			sb.append(x);
			sb.append(' ');
		}
		sb.append(']');
		return sb.toString();
	}
}
```

4. Consider symbolically executing the following code segment:

```
 1:  static void m(int x, int y) {
 2:      if (x < y) {
 3:          x++;
 4:          if (x + 1 < y - 1) {
 5:              y--;
 6:              if (x + 2 < y - 2) {
 7:                  System.out.println("got here!");
 8:              }
 9:          }
10:      }
11:  }
```

(a) List all the paths explored by forward symbolic execution (using the given line numbers).

(b) State the path condition for the path that reaches the `println` invocation.