```
/**
 * StackMachineInstruction.java
 * A generic instruction for an arithmetic stack machine.
 *
 * @author Hawk Weisman
 * @see StackMachine
 * @see Stack
 *
 * PLEDGE:
 */

public class StackMachineInstruction {

    public enum InstructionType {       // InstructionTypes are the possible
        identities of an instruction
        ADD, SUBTRACT, DIVIDE, MULTIPLY, MODULO, SWAP, OPERAND
    }

    private int value;              // if this is an operand, stores its value.
                                    // zero if it's an operator, so that it evals
                                        quasi-correctly
    private InstructionType type;   // the type identity of this instruction

    /**
     * 1-param constructor (operator)
     * @param type the type identity of this operator
     */
    public StackMachineInstruction (InstructionType type) {
        this.type = type;
        value = 0;
    }

    /**
     * 1-param constructor (operand)
     * @param value the value of this operand
     *
     * since any StackMachineInstruction with a non-zero value
     * is an operand, the type identity is set to operand by default
     */
    public StackMachineInstruction (int value) {
        this.type = InstructionType.OPERAND;
        this.value = value;
    }

    /**
     * Returns the type of this instruction
     * @return type the type of this instruction
     */
    public InstructionType getType () {
        return type;
    }

    /**
     * Returns the value of this instruction (if it is an operand)
     * @return value the value of this instruction (null if it is not an operand)
     */
```

```
public int getValue () {
    if (type == InstructionType.OPERAND) {
        return value;
    } else {
            return 0;
    }
}

/**
 * Evaluates this instruction in the context of stack s.
 * @param s The StackMachineInstruction stack of which this instruction is
     the top.
 */
public void eval (Stack<StackMachineInstruction> s) throws FullStackException
    {
    int result;
    switch (type) {
        case ADD:
            result = s.pop().getValue() + s.pop().getValue();
            s.push(new StackMachineInstruction(result));
            break;
        case SUBTRACT:
            result = s.pop().getValue() - s.pop().getValue();
            s.push(new StackMachineInstruction(result));
            break;
        case MULTIPLY:
            result = s.pop().getValue() * s.pop().getValue();
            s.push(new StackMachineInstruction(result));
            break;
        case DIVIDE:
            result = s.pop().getValue() / s.pop().getValue();
            s.push(new StackMachineInstruction(result));
            break;
        case MODULO:
            result = s.pop().getValue() % s.pop().getValue();
            s.push(new StackMachineInstruction(result));
            break;
        case SWAP:
            s.swap();
            break;
        default:
            s.push (new StackMachineInstruction(this.getValue()));
            break;
    }
}

/**
 * Returns a String representing the state of this instruction
 * @return a string representing the state of this instruction
 */
public String toString () {
    switch (type) {
        case ADD:
            return "+";
        case SUBTRACT:
            return "-";
```

```
            case MULTIPLY:
                return "*";
            case DIVIDE:
                return "/";
            case MODULO:
                return "%";
            case SWAP:
                return "s";
            case OPERAND:
                return "" + value;
            default:
                return "" + value;
            }
        }
}
```