Static Analysis Report

Colton Shoenberger

I used FindBugs, as well as the SolarLint plug-in for Eclipse, to analyze my program.

The following report summarizes a list of warnings I received from these tools, as well as changes I made to address them.

FindBugs Warnings

1. “Reliance on default encoding”

Reliance on default encoding

Found a call to a method which will perform a byte to String (or String to byte) conversion, and will assume that the default platform encoding is suitable. This will cause the application behaviour to vary between platforms. Use an alternative API and specify a charset name or Charset object explicitly.

Bug kind and pattern: Dm - DM_DEFAULT_ENCODING

I received this message in response to these lines of my original code:

```
Scanner scan = new Scanner(System.in);

[…]

System.setIn(new ByteArrayInputStream(test1.getBytes()));
```

These were rewritten to specify use of the UTF-8 charset:

```
Scanner scan = new Scanner(System.in, "UTF-8");

[…]

System.setIn(new ByteArrayInputStream(
      test1.getBytes(StandardCharsets.UTF_8)));
```

2. **“Method concatenates strings using + in a loop”**

Method concatenates strings using + in a loop

The method seems to be building a String using concatenation in a loop. In each iteration, the String is converted to a StringBuffer/StringBuilder, appended to, and converted back to a String. This can lead to a cost quadratic in the number of iterations, as the growing string is recopied in each iteration.

Better performance can be obtained by using a StringBuffer (or StringBuilder in Java 1.5) explicitly.

[…]

Bug kind and pattern: SBSC - SBSC_USE_STRINGBUFFER_CONCATENATION

I received this message in response to these lines of my original code:

```
String legalMoves = “”;
[…]
for (int x = 0; x < 8; x++) {
      for (int y = 0; y < 8; y++) {
            […]
            if […] {
                  legalMoves += toColumn(x) + "" + toRow(y) + " ";
                  numMoves++;
            }
      }
}
```

I initially rewrote this using a StringBuffer, as suggested. However, this then generated another warning from SolarLint, which advised me to use a StringBuilder instead, so I ended up doing that. This also was an opportunity for me how to consider how to properly place commas in the resulting string:

```
String legalMoves = "";
String prefix = "";
StringBuilder build = new StringBuilder();
[…]
for (int x = 0; x < 8; x++) {
      for (int y = 0; y < 8; y++) {
            […]
            if […] {
            build.append(prefix);
            prefix = ", ";
            build.append(toColumn(x) + “” + toRow(y));
            numMoves++;
            }
      }
}
legalMoves = build.toString();
```

3. “Consider returning a zero length array rather than null”

Consider returning a zero length array rather than null

It is often a better design to return a length zero array rather than a null reference to indicate that there are no results (i.e., an empty list of results). This way, no explicit check for null is needed by clients of the method.

On the other hand, using null to indicate "there is no answer to this question" is probably appropriate. For example, File.listFiles() returns an empty list if given a directory containing no files, and returns null if the file is not a directory.

Bug kind and pattern: PZLA - PZLA_PREFER_ZERO_LENGTH_ARRAYS

This error was in response to the findKing method, which searches for the location of the king on the board. If the King piece was not found, it originally returned null. The original code was:

```
for (int x = 0; x < 8; x++) {
      for (int y = 0; y < 8; y++) {
            Piece piece = board[y][x];

            if (piece instanceof King && piece.white == white) {
                  Return [x, y]
            }
      }
}
return null;
```

The good news about this error is that the oneKing method, which is called during user input, will prevent this method from ever having to return an empty array, because each team must have exactly one king placed on the board. Despite that, I still adjusted the code so that it would return an array either way.

```
int[] location = new int[2];

for (int x = 0; x < 8; x++) {
      for (int y = 0; y < 8; y++) {
            Piece piece = board[y][x];

            if (piece instanceof King && piece.white == white) {
                  location[0] = x;
                  location[1] = y;
            }
      }
}
return location;
```

SolarLint Warnings

4. “Replace this if-then-else statement by a single return statement”

I received this message in response to the legalMove method for all six of my pieces. Initially I was using if-then-else statements with multiple return options. This is an example of the original code I wrote to validate legal moves for the Queen piece:

```
boolean legalMove(int xFrom, int yFrom, int xTo, int yTo, boolean
      capture) {

      // Queen can move in a straight line along a row or column
      // (dx = 0 or dy = 0)
      if ((xFrom == xTo && yFrom != yTo) ||
            (xFrom != xTo && yFrom == yTo)) {
                  return true;
      }

      // Queen can also move diagonally (dx = dy)
      else if (Math.abs(xFrom - xTo) == Math.abs(yFrom - yTo)){
                  return true;
      }

      else {
                  return false;
      }
```

For each piece, I rewrote the method to provide a single return statement. This is an example of how I rewrote the legalMove method for the Queen piece:

```
boolean legalMove(int xFrom, int yFrom, int xTo, int yTo, boolean
      capture) {

      // Queens can move in a straight line along a row or column
      // (dx = 0 or dy = 0)
      return (((xFrom == xTo && yFrom != yTo) ||
            (xFrom != xTo && yFrom == yTo)) ||

            // Queens can also move diagonally (dx = dy)
            (Math.abs(xFrom - xTo) == Math.abs(yFrom - yTo)));
      }
```

5. **<u>“Refactor this method to reduce it’s cognitive complexity from 16 to the 15 allowed.”</u>**

I received this message in response to my validFormat method, which has the task of validating that the user input is formatted properly. This is composed of several subtasks, which include:

(1) Confirming each word in the input is three or four characters
(2) Confirming if the word has four characters, the fourth character is a comma
(3) Confirming the first character in each word represents a valid type (i.e. K, Q, R, B, N, or P)
(4) Confirming the second character in each word represents a valid column (i.e. a-h)
(5) Confirming the third character in each word represents a valid row (i.e. 1-8)

Clearly, this is a lot of tasks for one method.
The code on page 6 shows how I originally wrote the method.
The code on pages 7-8 show how I rewrote the code, splitting it into five methods.

```
// ORIGINAL

// Method for checking the input to confirm the formatting is valid.
static boolean validFormat(String str) {

      String nextPiece;
      char type, column, row;
      Scanner scan = new Scanner(str);

      // Check the input to confirm it is valid
      while (scan.hasNext()) {

            // Assign each word to nextPiece
            nextPiece = scan.next();

            // If the word is less than three characters or more than four,
            // it is invalid
            if (nextPiece.length() < 3 || nextPiece.length() > 4) {
                  return false;
            }

            // If it is four characters, the last character should be a comma
            if (nextPiece.length() == 4 && nextPiece.charAt(3) != ',') {
                  return false;
            }

            // The first character of nextPiece states its type
            // Valid types are K, Q, R, B, N, P
            type = nextPiece.charAt(0);
            if (!(type == 'K' || type == 'Q' || type == 'R' || type == 'B' ||
                  type == 'N' || type == 'P')) {
                  return false;
            }

            // The second character of nextPiece states its column
            // Valid columns are a-h
            column = nextPiece.charAt(1);
            if (!(column == 'a' || column == 'b' || column == 'c' ||
                  column == 'd' || column == 'e' || column == 'f' ||
                  column == 'g' || column == 'h')) {
                  return false;
            }

            // The third character of nextPiece states its row
            // Valid rows are 1-8
            row = nextPiece.charAt(2);
            if (!(row == '1' || row == '2' || row == '3' || row == '4' ||
                  row == '5' || row == '6' || row == '7' || row == '8')){
                  return false;
            }
      }
      // If it passes all the above tests, return true
      return true;
}
```

```
// REWRITE (FINAL CODE)

// Method for checking the input to confirm the formatting is valid.
static boolean validFormat(String str) {

	String nextPiece;
	char type = 0;
	char column = 0;
	char row = 0;
	Scanner scan = new Scanner(str);

	// Check the input to confirm it is valid
	while (scan.hasNext()) {

		// Assign each word to nextPiece
		nextPiece = scan.next();

		// If nextPiece is of valid length, assign the first three
		// characters to type, column, and row
		if (validLength(nextPiece)) {
			type = nextPiece.charAt(0);
			column = nextPiece.charAt(1);
			row = nextPiece.charAt(2);
		}

		// If nextPiece is not of valid length,
		// or one of the characters is not proper, return false
		if (!validLength(nextPiece) || !validType(type) ||
			!validColumn(column) || !validRow(row)) {
			scan.close();
			return false;
		}
	}
	// If it passes all the above tests, return true
	scan.close();
	return true;
}

// Method to validate each word in the string is the proper length
static boolean validLength(String str) {
	return (str.length() == 3 ||
		(str.length() == 4 && str.charAt(3) == ','));
}

// Method to validate the first character properly specifies the type
static boolean validType(char type) {
	return (type == 'K' || type == 'Q' || type == 'R' || type == 'B' ||
		type == 'N' || type == 'P');
}

// Method to validate the second character properly specifies the column
static boolean validColumn(char column) {
	return (column == 'a' || column == 'b' || column == 'c' ||
		column == 'd' || column == 'e' || column == 'f' ||
		column == 'g' || column == 'h');
}
```

```
// Method to validate the third character properly specifies the piece's row
static boolean validRow(char row) {
      return (row == '1' || row == '2' || row == '3' || row == '4' ||
            row == '5' || row == '6' || row == '7' || row == '8');
}
```

6. Miscellaneous

To conclude this report, I will mention some other minor warnings I received and adjustments I made to correct them. I will also mention some suggestions the SolarLint tool provided me that I deemed trivial and chose not to address.

- **"Dead store to local variable" (FindBugs) / "Remove this useless assignment to local variable" (SolarLint)**
  I received these errors a few times as I was rewriting code, indicating that I had assigned a value to a local variable that I did not end up using. This generally resulted from sloppy editing and failing to clean up after myself. In these cases, I was able to simply remove the variable that triggered the warning, as it was unneeded.
- **"Remove the declaration of thrown exception 'java.io.IOException', as it cannot be thrown from the methods body"**
  Initially, all system input and output was handled in the main method, and I could not determine a way to build unit tests for the main method. As I was writing the userInput method (and testUserInput tests), I was at some point informed that I must throw the IOException. I spent a while rewriting this method, and once I had finalized it, the exception apparently no longer needed to be declared

Warnings I chose not to address:

- **Define a constant instead of duplicating this literal ["..." x] times**
  This message resulted from reuse of several string literals in Chess.java and ChessTest.java. None of these strings were repeated more than 4 times. I believe that declaring a constant for these strings would have a negative impact on readability, and would not produce any meaningful improvement in performance.
- **Declare "..." on a separate line**
  SolarLint's rule description here argues that declaring multiple variables on one line is difficult to read. In my program, the largest number of variables declared on one line is three. I felt that separating these onto multiple lines would have made the code unnecessarily long.
- **Replace this use of System.out or System.err by a logger**
  SolarLint displays this as a high priority warning, so I spent a lot of time trying to figure this one out. I spent a while reading about the Java Logging API and messed around with log4j2 several hours as well. Ultimately, for the purpose of this program, this felt like more trouble than it was worth. The only annoying side effect of the System.out.print calls that I was experiencing was that I was getting unnecessary output to the console when running tests in Maven. I was able to resolve this by using System.setOut and creating a dummy output.
- **Use try-with-resources or close this "Scanner" in a "finally" clause.**
  This warning resulted from two methods in my program where a call to close the scanner is made a several locations—sometimes within loops, but in every case just before the method returns a value. Thus, each scanner is guaranteed to close before the method exits, which is why I did not feel the need to heed this suggestion.