Debugging

Table of contents

Practical for week 5: using the IntelliJ debugger to find and fix bugs in a small provided codebase of chat bots (`ChatBot` implementations) — reinforcing java-junit[1] (the provided tests are what reveal each bug) and java-exceptions[2] (`RecursiveFibonacci`'s bug manifests as a `StackOverflowError`).

This practical is hands-on IntelliJ-debugger work rather than a computable Q&A, so there are no interactive checks here — just each bug and its fix, as a static reference/answer key.

Setup: a provided `.zip` (from Blackboard) contains `ChatBot.java` (the shared interface) plus several bot implementations and their JUnit tests, to be placed under a `debugging/` package in `src//test/` source roots. Running the tests initially shows 37 tests total, 24 failing. **Ground rule**: no modifying code (implementation *or* tests) until the bug is actually found via the debugger — no debug-print-statement shortcuts.

```
public interface ChatBot {
    String replyTo(String username, String message);
}
```

[1] courses/csse2002/java-junit.pdf
[2] courses/csse2002/java-exceptions.pdf

Bug 1 — `MockBot`

`MockBot` should reply to any message with the message repeated in *alternating caps* — letters up to and including `'O'` lowercase, letters after `'O'` uppercase:

```
private String mock(String message) {
    StringBuilder result = new StringBuilder();
    for (char letter : message.toCharArray()) {
        if (letter > 'O') {
            result.append(Character.toUpperCase(letter));
        } else {
            result.append(Character.toLowerCase(letter));
        }
    }
    return result.toString();
}
```

> **Working**
>
> The bug is that the *original* message's case is never normalised first — `letter > 'O'` compares the character's existing case-sensitive code point directly, so any already-lowercase letter after `'o'` (lowercase) in the alphabet compares incorrectly against the uppercase `'O'` boundary. Fixing it means normalising the message to uppercase before iterating, so the comparison is always against consistent-case letters:
>
> ```
> private String mock(String message) {
>     StringBuilder result = new StringBuilder();
>     for (char letter : message.toUpperCase().toCharArray()) {
>         if (letter > 'O') {
>             result.append(Character.toUpperCase(letter));
>         } else {
>             result.append(Character.toLowerCase(letter));
>         }
>     }
>     return result.toString();
> }
> ```

Bug 2 & 3 — `GuessingGameBot`

A bot that replies "yes"/"no" to `guess higher <n>` / `guess lower <n>` against a secret number, and "you win!"/"you lose!" for `guess equal <n>`. Two independent bugs:

Bug 2 — message splitting. `split(String)` breaks a message into words on spaces:

```
protected static List<String> split(String input) {
    List<String> output = new ArrayList<>();
    int index = 0, newIndex;
    while (index != -1) {
        newIndex = input.indexOf(' ', index + 1);
        if (newIndex == -1) {
            output.add(input.substring(index));
        } else {
            output.add(input.substring(index, newIndex));
        }
        index = newIndex;
    }
    return output;
}
```

Bug 3 — comparison logic. The `higher`/`lower` cases compare the wrong way:

```
case "higher" -> {
    int guess = Integer.parseInt(words.get(2));
    return guess > SECRET_NUMBER ? "yes" : "no";
}
case "lower" -> {
    int guess = Integer.parseInt(words.get(2));
    return guess < SECRET_NUMBER ? "yes" : "no";
}
```

Working

Bug 2 fix: `substring(index)` *includes* the character at `index` itself, but after finding a space, `index` is reassigned directly to that space's position (`newIndex`) — so the next word's substring starts with a leading space. Fix: advance past the space (`newIndex + 1`), and `break` once the last word has been added (since `index` would otherwise become `-1` from `newIndex`, which is *already* handled by the `while` condition, but the assignment ordering still needs correcting):

```
while (index != -1) {
    newIndex = input.indexOf(' ', index + 1);
    if (newIndex == -1) {
        output.add(input.substring(index));
        break;
    } else {
        output.add(input.substring(index, newIndex));
    }
    index = newIndex + 1;
}
```

Bug 3 fix: the `>`/`<` are swapped — a `"guess higher"` should return `"yes"` when the guess is *lower* than the secret (i.e. the secret is higher than the guess), and vice versa:

```
case "higher" -> {
    int guess = Integer.parseInt(words.get(2));
    return guess < SECRET_NUMBER ? "yes" : "no";
}
case "lower" -> {
    int guess = Integer.parseInt(words.get(2));
    return guess > SECRET_NUMBER ? "yes" : "no";
}
```

Bug 4 & 5 — `MultiplayerGuessingGameBot`

Extends `GuessingGameBot` to let multiple users each submit one guess, with the game's host finishing it to reveal a ranked leaderboard. Two independent bugs:

Bug 4 — host check inverted:

```
if (message.equals("finish game")) {
    if (username.equals(host)) {
        return "You do not have a game running.\n" +
                "Host must finish game.";
    }
    return results();
}
```

Bug 5 — leaderboard rank off-by-one:

```
private String results() {
    // ...
    int rank = 0;
    for (Map.Entry<String, Integer> entry : entries) {
        builder.append("\n").append(rank).append(") ")...
        rank++;
    }
    return builder.toString();
}
```

Working

Bug 4 fix: the condition is backwards — a *non-host* finishing the game should get the "you do not have a game running" message, not the host:

```
if (!username.equals(host)) {
    return "You do not have a game running.\n" +
            "Host must finish game.";
}
return results();
```

Bug 5 fix: rankings should start at 1, not 0:

```
int rank = 1;
```

Bug 6 & 7 — `CommitBot`

Lets a user generate a random commit message (`commit new`), save the last-generated one (`commit save`), and reload it later (`commit load`). Two independent bugs:

Bug 6 — generated message never recorded as "last":

```
if (message.endsWith("new")) {
    try {
        String commit = generateMessage();
        return commit;
    } catch (IOException | URISyntaxException e) {
        return "IOException trying to fetch commit message";
    }
}
```

Bug 7 — `load` reads from the wrong map:

```
if (message.endsWith("load")) {
    if (!lastMessages.containsKey(username)) {
        return "No saved message to load";
    }
    return lastMessages.get(username);
}
```

Working

Bug 6 fix: `commit new` never stores the generated message into `lastMessages`, so `commit save` (which reads from `lastMessages`) always sees a stale (or missing) value:

```
String commit = generateMessage();
lastMessages.put(username, commit);
return commit;
```

Bug 7 fix: `commit load` should read the *saved* message, not the *last generated* one — otherwise `load` just repeats whatever was most recently generated, ignoring `commit save` entirely:

```
if (message.endsWith("load")) {
    if (!savedMessages.containsKey(username)) {
        return "No saved message to load";
    }
    return savedMessages.get(username);
}
```

Bug 8 — `RecursiveFibonacci` (bonus)

Tests throw a `StackOverflowError` — a recursive method's base case is either missing or unreachable, so it recurses forever (until the call stack runs out of memory).

Working

The recursive `calculate(n, x)` only has a base case for `n == 1`. When `x > 1`, the recursive subtraction steps by more than 1 at a time, so `n` can skip straight past `1` into negative numbers — `n == 1` is never reached, and the recursion never terminates.
Two possible fixes (given $\mathcal{F}(1) = \mathcal{F}(2) = 1$):

```
// Preferred: widen the existing base case so it can't be skipped over.
public static int calculate(int n, int x) {
    if (n <= 2) {
        return 1;
    }
    // ...
}
```

```
// Alternative: add an explicit n == 0 (or n < 1) base case.
public static int calculate(int n, int x) {
    if (n == 0) { // or n < 1
        return 0;
    }
    // ...
}
```

The first (`n <= 2`) is preferred: the second approach still lets recursive calls that violate the method's own precondition (`n >= 1`) run to completion instead of eliminating them.

Bug 9 — `CachedFibonacci` (challenge)

Gives incorrect results for larger Fibonacci numbers. The cache moves the most-recently-used entry to the front of two parallel arrays (`cacheKeys`/`cacheValues`) each lookup:

```
int key = cacheKeys[index], value = cacheValues[index];
for (int i = index - 1; i >= 0; --i) {
    cacheKeys[i + 1] = cacheKeys[i];
    cacheValues[i + 1] = cacheValues[i];
}
cacheKeys[0] = key;
cacheValues[0] = value;
```

i Working

After shuffling the found entry to the front of the cache, `index` (the variable tracking *where* the entry was found) is never updated to reflect its new position (`0`) — so subsequent logic that relies on `index` still thinks the entry lives at its old position:

```
cacheKeys[0] = key;
cacheValues[0] = value;
index = 0;
```