Debugging

Table of contents

Setup and ground rules 1
Bug 1 — `MockBot` 2
Bug 2 & 3 — `GuessingGameBot` 4
Bug 4 & 5 — `MultiplayerGuessingGameBot` 7
Bug 6 & 7 — `CommitBot` 9
Bug 8 — `RecursiveFibonacci` (bonus) 11
Bug 9 — `CachedFibonacci` (challenge) 12

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 and ground rules

A provided `.zip` (from Blackboard) contains thirteen files — `ChatBot.java` (the shared interface), five bot implementations, and a JUnit test class for each:

```
CachedFibonacci.java
CachedFibonacciTest.java
ChatBot.java
CommitBot.java
CommitBotTest.java
GuessingGameBot.java
GuessingGameBotTest.java
MockBot.java
MockBotTest.java
MultiplayerGuessingGameBot.java
MultiplayerGuessingGameBotTest.java
RecursiveFibonacci.java
RecursiveFibonacciTest.java
```

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

The setup is itself part of the exercise: create an IntelliJ project with a `src/` source root (blue) and a `test/` test source root (green), read the top of the files to discover they all declare a `debugging` package, create `debugging/` under **both** roots, and copy the files in accordingly. If it worked, the run shows **37 tests, 24 of them failing**.

This is also a Git exercise (java-version-control[3]): initialise a repository before you start and commit after **each** bug fix with a message describing the fix, so the history is one commit per defect.

> **!** Ground rules
>
> The practical is about learning the debugger, so for this practical only, modifying the code to add debugging print statements is **banned**. Modifying the code at all is **banned** until you have found the bug and are ready to start fixing it. The ban applies to the tests as well.

For the number-guessing bot the work splits into three separate tasks, and they are the actual method being taught — the fix is the last step, not the first:

1. **Where** does the code stop working?
2. **Why** does the code stop working?
3. **How** can you fix the code so that it does work?

Every other bot gets the same sequence in compressed form: step through the failing tests with the debugger to work out why they fail, then implement the fix, then repeat until the tests pass.

Every bot implements one interface, so a failing test always points at one `replyTo` call:

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

The method takes the user's username and their message, and either replies or returns `null` to indicate no response.

Bug 1 — `MockBot`

`MockBot` responds to all messages with a copy of the message in alternating caps:

```
[you]> its a mock bot world after all iTS a mock boT WoRld afTeR all
```

[3] courses/csse2002/java-version-control.pdf

```java
ChatBot bot = new MockBot();
System.out.println(bot.replyTo("you", "its a mock bot world after all"));
// iTS a mock boT WoRld afTeR all
```

The rule is not "alternate letter by letter" but **a threshold on the letter**: once the message has been folded to uppercase — which is exactly what the fix adds — a letter at or below `'O'` comes out lowercase and a letter above `'O'` comes out uppercase. (Check it against the example: `a`, `b`, `c`, `d`, `e`, `f`, `i`, `k`, `l`, `m`, `o` all sit at or below `O` in the alphabet and are printed lowercase, while `r`, `s`, `t`, `w` sit above it and are printed `R`, `S`, `T`, `W`. The listing below is the **buggy** body, i.e. before that fold — run as it stands it compares the raw lowercase codes instead, which is the bug.)

```java
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 comes from improper handling of lowercase letters: all lowercase letters are converted to capitals. Converting the message to uppercase first fixes it.

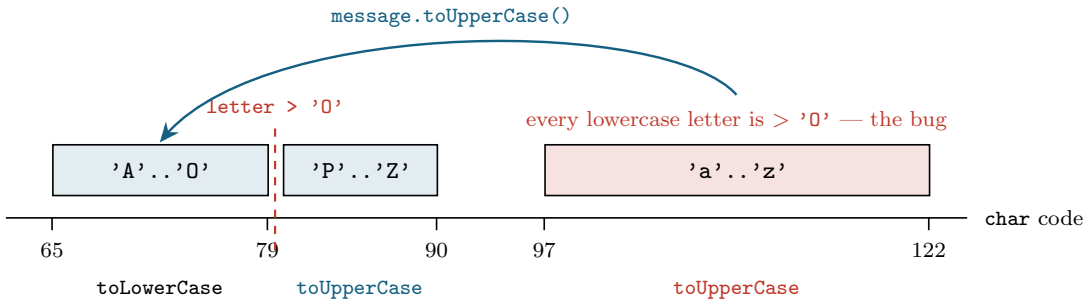

Figure 1: Why every lowercase letter comes out shouted, and what toUpperCase fixes. Comparing a char with > compares its numeric code, and the entire lowercase alphabet (97-122) sits above the threshold letter O (79), so the buggy loop routes all of a-z into the toUpperCase branch. Folding the message to uppercase first collapses a-z onto A-Z, putting each letter back on the side of the threshold its position in the alphabet intends.

Comparing two `char`s with `>` compares their numeric codes, and Java's letter codes run `'A'`–`'Z'` = 65–90 then `'a'`–`'z'` = 97–122. Every lowercase letter therefore exceeds `'O'` (79) regardless of where it sits in the alphabet, so the buggy loop sends the whole of `a`–`z` down the `toUpperCase` branch: every lowercase letter is converted to a capital.
Normalising the message to uppercase **before** the loop puts every letter into the 65–90 range, where the comparison against `'O'` means what it was meant to mean:

```
private String mock(String message) {
    ...
-   for (char letter : message.toCharArray()) {
+   for (char letter : message.toUpperCase().toCharArray()) {
    ...
}
```

The debugger route to this: set a breakpoint inside the loop, watch `letter` and the branch taken, and the first lowercase character shows the wrong branch being entered.

Bug 2 & 3 — `GuessingGameBot`

A bot that plays a number-guessing game: the user asks whether the secret "magic number" is higher or lower than a guess, and the bot answers.

```
[you]> help
Guessing game bot!
```

```
There are 3 options
1) guess higher <number>
2) guess lower <number>
3) guess equal <number>
[you]> guess higher 2
yes
[you]> guess lower 5
no
[you]> guess equal 3
you lose!
```

Two independent bugs.

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

```
private static List<String> split(String input) {
    ...
    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;
    }
}
```

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

```
public static String replyTo(String message) {
    ...
    if (type.equals("higher")) {
        ...
        return guess > SECRET_NUMBER ? "yes" : "no";
    } else if (type.equals("lower")) {
        ...
        return guess < SECRET_NUMBER ? "yes" : "no";
        ...
    } else if (type.equals("equal")) {
    ...
}
```

Working

Reading the documentation in the source files helps in working out how the code functions. Both bugs are only visible once you know what `split` and `replyTo` are each *supposed* to return.

Bug 2 fix. Messages are not split correctly, leaving extra whitespace before words, because `substring()` includes the char at the first index it is given.

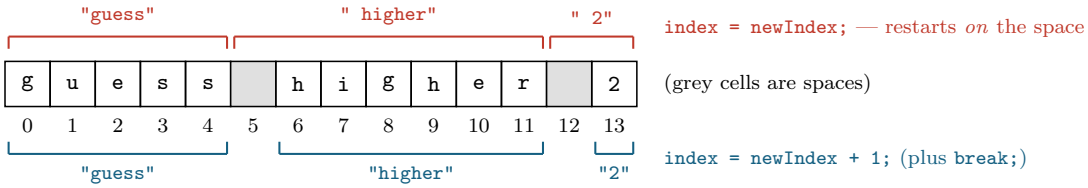

Figure 2: Bug 2 traced on the message the transcript uses. Top: the substring ranges the buggy loop produces, because index is reassigned to the position OF the space rather than one past it, so every word after the first keeps a leading space. Bottom: the ranges after the fix. The leading space is not cosmetic - it makes type.equals(higher) false and would make Integer.parseInt on the last token throw.

Stepping `"guess higher 2"` through the buggy loop gives `["guess", " higher", " 2"]` — the two later tokens carry a leading space, which is enough to make `type.equals("higher")` false and to make `Integer.parseInt(" 2")` throw a `NumberFormatException`. The fix advances past the space:

```
private static List<String> split(String input) {
    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;
+       index = newIndex + 1;
    }
}
```

The `break` is **not optional, and it is a consequence of the other line**: with `index = newIndex + 1`, the loop's own `while (index != -1)` guard can no longer stop it — on the last word `newIndex` is `-1`, so `index` becomes `0` and the loop restarts from the

beginning of the string forever. Adding the `+ 1` without the `break` trades a wrong answer for an infinite loop.
Bug 3 fix. Guesses are checked against the secret number incorrectly. Swapping the `>` and `<` fixes it.

```
public static String replyTo(String message) {
    ...
    if (type.equals("higher")) {
        ...
-       return guess > SECRET_NUMBER ? "yes" : "no";
+       return guess < SECRET_NUMBER ? "yes" : "no";
    } else if (type.equals("lower")) {
        ...
-       return guess < SECRET_NUMBER ? "yes" : "no";
+       return guess > SECRET_NUMBER ? "yes" : "no";
        ...
    } else if (type.equals("equal")) {
    ...
}
```

The reading that makes the swap obvious: `guess higher 2` asks *"is the **secret** higher than 2?"*, so the answer is `"yes"` exactly when `2 < SECRET_NUMBER`. Checked against the transcript above — `guess higher 2` → `yes` and `guess lower 5` → `no` — the fixed code says the secret is greater than 2 and not less than 5, which the `guess equal 3` → `you lose!` line is consistent with.

Bug 4 & 5 — `MultiplayerGuessingGameBot`

The multiplayer variant lets several users each submit **one** guess; whoever is closest wins, and only the host may finish the game:

```
[you]> start game
Starting new game...
[you]> guess 12
[root]> guess 5
[admin]> guess 100
[test]> guess 34
[root]> finish game
You do not have a game running.
Host must finish game.
[you]> finish game
The secret number was 13
```

```
1) you (12)
2) root (5)
3) test (34)
4) admin (100)
```

Two independent bugs.

Bug 4 — host check inverted:

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

Bug 5 — leaderboard rank off-by-one:

```
private String results() {
    ...
    int rank = 0;
    ...
}
```

Working

Bug 4 fix. Finishing a game is not handled correctly. The user finishing the game should be the same as the host; the current implementation does the opposite. The refusal message belongs to the **non**-host:

```
public String replyTo(String username, String message) {
    ...
-   if (username.equals(host)) {
+   if (!username.equals(host)) {
    ...
}
```

Read against the transcript: `[root]> finish game` is refused because `root` is not the host, and `[you]> finish game` succeeds because `you` started the game. The buggy version does exactly the reverse — the host is the only user who *cannot* end their own game.

Bug 5 fix. The leaderboard has an off-by-one error in the ranking: it starts from zero

instead of one.

```
private String results() {
    ...
-   int rank = 0;
+   int rank = 1;
    ...
}
```

The transcript's leaderboard begins at `1) you (12)`, so the counter that labels the rows is a display index rather than an array index — and Java's habit of counting positions from zero is what makes `int rank = 0;` look right at a glance.

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`):

```
[you]> commit new
I know what I am doing. Trust me.
[you]> commit save
Commit saved
[you]> commit new
Does this work
[you]> commit load
I know what I am doing. Trust me.
```

Two independent bugs. The transcript requires that `commit load` returns the message that was **saved**, not the one most recently generated.

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

```
public String replyTo(String username, String message) {
    ...
            String commit = generateMessage();
            return commit;
    ...
}
```

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

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

Working

Bug 6 fix. The last generated commit message is not recorded when it is generated. `commit save` reads whatever `lastMessages` holds for the user, so if `commit new` never writes there, `save` has nothing (or something stale) to promote:

```
public String replyTo(String username, String message) {
    ...
            String commit = generateMessage();
+           lastMessages.put(username, commit);
            return commit;
    ...
}
```

Bug 7 fix. The load command loads commit messages from the last message rather than the saved message.

```
public String replyTo(String username, String message) {
    ...
        if (message.endsWith("load")) {
-           if (!lastMessages.containsKey(username)) {
+           if (!savedMessages.containsKey(username)) {
                return "No saved message to load";
            }
-           return lastMessages.get(username);
+           return savedMessages.get(username);
        }
    ...
}
```

The two maps exist for two different lifetimes — `lastMessages` is *whatever was generated most recently*, `savedMessages` is *what the user chose to keep* — and each bug is one of them being used where the other belongs. The bugs mask each other, so the transcript is worth re-reading between fixes. Fix bug 6 alone and `commit load` starts returning the

most recently generated message — `"Does this work"` — rather than the saved one; fix bug 7 alone and `savedMessages` still has nothing in it to load. Only both fixes together reproduce the transcript's last line.

Bug 8 — `RecursiveFibonacci` (bonus)

The tests throw a "StackOverflow" exception. A stack overflow exception is thrown when a function recursively calls itself so many times that the program runs out of stack memory. A recursive method needs a base case so that the function knows when to stop. Two things to check: whether this code has a base case, and whether it is being reached.

i Working

The issue arises when `x > 1`: the recursive calls to `calculate` then include subtractions of 2 or greater, so the base case of `x == 1` is skipped, resulting in infinite calls of negative numbers. The base case is written `n == 1`, an **equality** test, so any step that moves `n` past 1 by more than one at a time misses it entirely and the recursion runs away into the negatives.
Two fixes, given $\mathcal{F}(1) = \mathcal{F}(2) = 1$:

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

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

The first is preferred. The second preserves and tolerates recursive calls that violate the precondition, `n >= 1`; the first removes the offending calls where `n < 1`. In the language of java-specification[4]: the second fix makes the method *cope* with arguments outside its own precondition, which quietly widens the contract; the first fix makes those calls never happen.

The general lesson is about the shape of a guard rather than this method: a base case tested with `==` is only safe if the recursion is guaranteed to land exactly on it. Testing with `<=` is robust to any step size.
A naming point: `StackOverflowError` is an `Error`, not an `Exception` — both are `Throwable`s, but `Error` marks conditions a program is not expected to catch and recover from. The distinction is drawn in java-exceptions[5]; calling it an exception is informal.

Bug 9 — `CachedFibonacci` (challenge)

"Code is broken. Please fix. Good luck." The symptom is that the results are wrong for **larger** Fibonacci numbers only — small inputs pass.

The cache is two parallel arrays, `cacheKeys` and `cacheValues`, kept in most-recently-used order: on a hit at position `index`, the entry is shuffled to the front.

```
...
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;
...
```

Working

The code gives incorrect results for larger Fibonacci numbers. The problem is that the index of the current number `n` is not updated after its position is shuffled to the front of the cache.

[5] courses/csse2002/java-specification.pdf

[5] courses/csse2002/java-exceptions.pdf

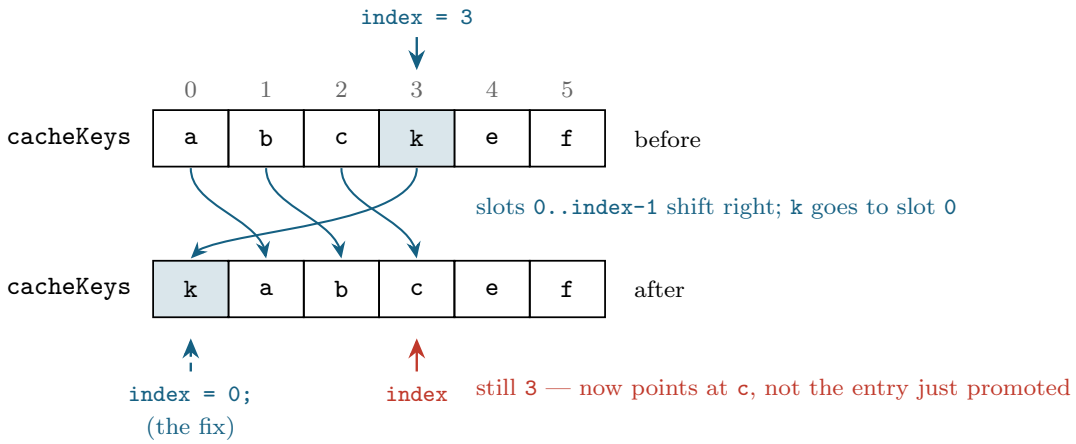

Figure 3: The move-to-front shuffle and the variable it forgets. The slot values a..f stand in for cache contents. Slots 0 through index-1 each shift one place right and the promoted key k is written into slot 0, so after the shuffle every entry at or below index has moved. index itself is untouched, so it now names a different entry than the one just looked up - which is why only lookups that hit deep in the cache, i.e. larger inputs, come back wrong.

```
...
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;
+ index = 0;
...
```

This bug is a good argument for the ground rule about the debugger. The symptom (*"wrong only for larger numbers"*) is the giveaway once the mechanism is on paper: `index == 0` is a hit at the front, where the shuffle moves nothing and the stale value is accidentally correct. Only a hit at `index > 0` — which needs a cache deep enough to have been populated, i.e. a larger input — leaves `index` naming the wrong slot. Reading the code alone, the missing line is invisible because nothing is *written* incorrectly; a variable is simply left behind by a state change happening around it.