

# CS15 Debugging Guide

## Table of Contents

[Introduction](#)

[General Tips & Tricks](#)

[Debugging Techniques](#)

[Strings & Printlines](#)

[Commenting out Code](#)

[Types of Bugs](#)

[Compiler Errors](#)

[Runtime Errors](#)

[Conclusion](#)

## Introduction

Creating any program is a three step process: designing, developing, and debugging.

In the **design** phase, you decide which features are needed and how you will organize the classes that implement these features. The majority of the coding happens in the **development** phase. Once you have testable code, you make sure it's working correctly in the **debugging** phase.

This process is definitely not linear; often you will have to change your design after you begin coding, and you should always be testing your code during development.

Most of the time spent writing any program is in the debugging phase. A well thought-out and thorough design will make development easier, and careful coding will simplify debugging tremendously. Nevertheless, every application is going to have bugs, so being able to debug is an essential skill.

This guide will go over common debugging methods, types of bugs you may (and most likely will!) come across throughout the course, and ways to find them. Feel free to reference this at any point during the semester, as it may come in handy! Before bringing a bug to TA hours, we expect you to try debugging it yourself first. Of course, everyone runs into tricky bugs, and your TAs are more than happy to help you once you've put in some thought. Bugs might seem very daunting at first, but the process becomes easier as you get better at debugging!

## General Tips & Tricks

1. Keep your code neatly [formatted](#)! This will make it easier for you to read over it when you run into a problem. Inline comments, well-organized methods, informatively-named variables, and consistent spacing are your friends.
2. Plan your design before you start coding. It's easy to make mistakes when editing code, especially when making big structural changes. A good initial design will help avoid this!
3. Code incrementally. When you run into a bug, figure out what's wrong before moving on. Otherwise, errors can stack on top of each other, making it more difficult to debug down the road.
4. Don't be afraid to delete code! This might feel like a step backward, but sometimes this can be the easiest way to make progress. If you think you might need the code again later, comment it out.

## Debugging Techniques

### Strings and Printlines

Before talking more about specific types of bugs, we're going to go over printlines, which will be helpful until you get access to more powerful means of debugging. Just like it is important to code incrementally, when there is a bug in your program, it is important to debug incrementally. You can do all of this using a built-in Java method, `System.out.println(<whatever you want to print>)`. This will print to your terminal whatever is inside the parentheses.

`System.out.println()` automatically calls a method `toString()` on any object being printed. Every Java object has one built in (although sometimes the output looks a little scary), so you won't have to write your own. Therefore, the following lines are synonymous:

```
System.out.println(_car.toString());  
System.out.println(_car);    //Java automatically calls toString()
```

**Output:**

```
>> Car@2d1ee34f  
>> Car@2d1ee34f
```

You can **concatenate** (meaning combine) strings using the `+` operator.

```
String name = "Andy";  
System.out.println("My name is " + name + "!");
```

**Output:**

```
>> My name is Andy!
```

By using a variable, `name`, the output of the `println` will vary based upon the value of the variable at the time the line is called. Because all objects have a built-in `toString()` method, this variable could be any type (`Strings`, `ints`, and many others).

You may be asking yourself, but why are `println`s useful? Let's say you run into a null pointer exception (more information on that below!), and are not sure where in your program that is happening. You could insert `System.out.println(variable == null)` throughout your code to find the point of error. Or, say that you have an array that keeps going out of bounds. You could use `println`s to track the variables within each loop to find where the error is happening. If you don't quite understand the value of `println`s just yet, don't worry! You'll catch on quickly as the course progresses!

### **\*\*Notes About Printline Style\*\***

1. Please make sure to **remove all printlines before handing in a project**. It is bad practice to leave `println`s in your code, and you will lose points!
2. When you are printing the value of a variable, it is generally a good idea to include a descriptive text with it so that when you look at your console, the `println`s have context.

For example:

Don't print

```
System.out.println(_name);
```

Do print

```
System.out.println("Professor's name: " + _name);
```

## Commenting out Code

Another useful debugging tool is commenting out code. You've already seen comments used to explain what the code does, but you can also use comments to temporarily disable lines of code. Remember that anything that comes after `//` on a line is a comment and will not be executed.

***In Atom and Eclipse, you can highlight a line or block of code and press CTRL + "/" to comment out or uncomment that code.***

Let's say we want to test our `JimHalpert` class method `pullPrank(...)`. We realize that we have an error, but are not sure where it's coming from. In order to locate the source, we comment out the method `stealStapler()` to make sure the other two methods being called are working properly up to that point.

```
1    public void pullPrank() {
2        this.dressLikeDwight();
3        this.buildPencilFence();
4        // this.stealStapler();
```

```
5      }
```

We find that if we run this code with line 4 disabled, everything works as intended, indicating our method `stealStapler()` is responsible for whatever is going wrong. Commenting out these lines lets us code `pullPrank(...)` incrementally, making sure that `dressLikeDwight()` and `buildPencilFence()` both work before moving on to `stealStapler()`.

**\*\*Note:** If working in Eclipse, you may notice the Eclipse Debugger tool on the top bar. This tool works by setting breakpoints to step through and examine your code as it runs, a skill you'll learn in greater detail if you take upper level CS courses. This is not widely used in CS15 as it can be very tricky to navigate, so we do not encourage you to rely on this for debugging. However, feel to play around with it sometime if you'd like!

## Types of Bugs

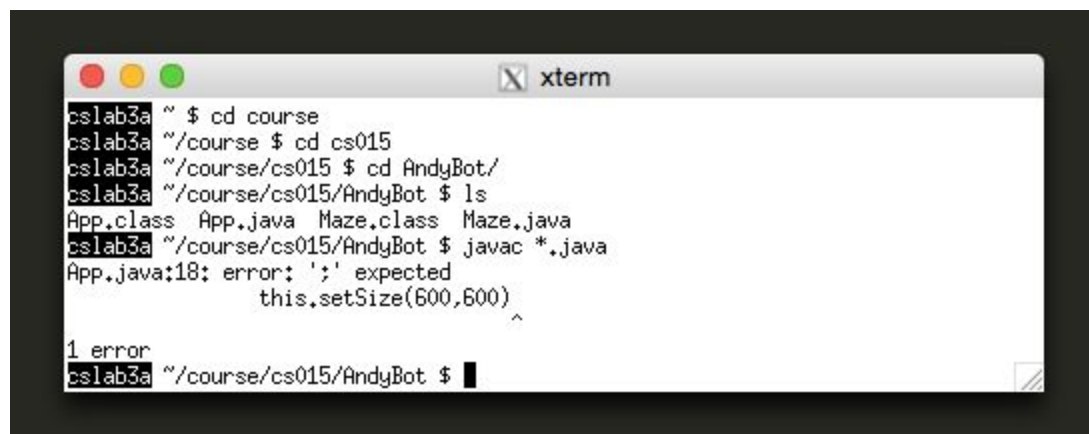
Let's talk about coding bugs. All Java bugs can be broken down into two categories: compile time bugs and runtime bugs.

### Compile Time Bugs

These happen when you violate the rules of the programming language and the compiler can't determine what you are trying to do. Examples of this would be: missing semicolon, a typo on a variable name, or calling a method that doesn't exist. Make sure you're familiar with Java syntax to help you avoid these errors!

### Finding Compile Time Errors

When you first run a program, you will need to move into the correct directory and then type "javac \*.java" into the terminal. This will compile all files ending with ".java". If you have a compile time error, you will get something which looks like this:

A screenshot of a terminal window titled 'xterm'. The terminal shows a series of commands and their outputs. The user navigates to the directory ~/course/cs015/AndyBot and runs 'javac \*.java'. The output shows a compilation error in App.java at line 18: 'error: ';' expected' followed by the code snippet 'this.setSize(600,600)'. The terminal also shows a list of files in the directory: App.class, App.java, Maze.class, and Maze.java. At the bottom, it says '1 error' and the prompt returns to the user.

```
cslab3a ~ $ cd course
cslab3a ~/course $ cd cs015
cslab3a ~/course/cs015 $ cd AndyBot/
cslab3a ~/course/cs015/AndyBot $ ls
App.class App.java Maze.class Maze.java
cslab3a ~/course/cs015/AndyBot $ javac *.java
App.java:18: error: ';' expected
    this.setSize(600,600)
                        ^
1 error
cslab3a ~/course/cs015/AndyBot $
```

This tells you a lot about the error that was found:

- The error occurred on line “18” in the “App” class
- The error is “ ‘;’ expected ”
- The compiler believes the error is at the end of line 18 - after “ (600,600) ”

## Common Compile Time Errors

- ‘Punctuation’ Errors: Note that these are not the only errors you might get from incorrect syntax, just some common ones. Examples may include:
  - error: ‘;’ expected
  - error: ‘)’ expected
  - <identifier> expected
 \*\*Tips to find/resolve bug\*\*
  - The java compiler will give you the class name and line number where it found an error. Check over the syntax for this line especially carefully!
  - If you have multiple errors, it is generally a good idea to start with the first one, because sometimes an earlier error will lead to the compiler thinking something later (which is correct) is also an error.
  - If you get 20 or 30 errors, don’t worry! You most likely forgot a bracket or parenthesis.
  - Note that the compiler’s suggestion for what should be fixed is not always correct! For example, sometimes it will say “ ‘;’ expected ” when actually you forgot a “=”.
- Method or Variable Errors: These are generally caused by using a method or variable name which is incorrect. Examples may include:
  - Error: cannot find symbol
 \*\*Tips to find/resolve bug\*\*
  - Check that all of the methods and variables you are using are spelled correctly, including capitalization!
  - Check that variables and methods have been declared correctly.

## Runtime Bugs

These indicate an error in the logic of the code. This can be confusing because the compiler tells you your code is okay, yet you still get errors! This is because the compiler only reads the code but doesn’t actually understand it, so non-syntax errors might not surface until the code is run. These bugs can either cause your program to crash (ex: `NullPointerException`) or cause incorrect functionality (ex: I click on one spot on the screen but the `Lite Peg` shows up in another), and may produce very long error messages. As time goes on, most of your time will be spent on runtime bugs, some of which can be extremely tricky.

## Finding Runtime Errors

Once your program will compile, it is time to run it. Type “`java <program name>.App`”. If you have a runtime error, something like this will print to your console:

```
bash-3.2$ java ratty.App
Exception in thread "main" java.lang.NullPointerException
    at ratty.RattyMenu.createTodaysDate(RattyMenu.java:51)
    at ratty.RattyMenu.<init>(RattyMenu.java:42)
    at ratty.App.<init>(App.java:42)
    at ratty.App.main(Hpp.java:47)
bash-3.2$
```

Exception Name

Stack Trace

package.Class      Method      File:Line#

1. **Stack Trace:** The list of all the methods executing when the error occurred. The top-most method is the one that was called most recently. **Start from the top and look for the first method that you wrote**, as the error probably occurred there. In a typical stack trace, there will be many methods that you did not write near the bottom of the list (they are either standard Java methods or CS15 support code methods) and your methods are usually near the top. *Support code and Java are both well tested and bug free*, so you should begin tackling the stack trace when it first mentions one of your methods.
2. **Exception Name:** The error that occurred. This will usually give you an idea of what might have happened and how to fix it. (A little about Java terminology: when a runtime error occurs, it is described as Java "throwing an exception".)
3. **Package.Class:** The location where the error occurred.
4. **Method:** The method where the error occurred. **NOTE:** `<init>` refers to the constructor.
5. **File:Line #:** The name of the file and the line number where Java thinks the error occurred. It's a good idea to go to the line the compiler is referring to begin debugging.

**NOTE:** The class or method that ran into the error is not necessarily the one that caused it. This happens most often if the method accepts a parameter. For example, if you passed a null value as the actual parameter into a method, an error would result because a null value should never have been passed. You can, however, trace down from this class or method to figure out where your error was caused.

There are many different Runtime Errors. For now, we're only going to introduce you to two, which are probably the ones that you will encounter most frequently at this point.

### Common Runtime Errors

- **NullPointerException:** These occur when Java encounters a null reference when it doesn't expect one. This usually happens when you try to use something that hasn't been initialized or instantiated. For example:

```
String name; // name hasn't been initialized
System.out.println("Hello, " + name);
```

**\*\*Tips to find/resolve bug\*\***

- You should check that every variable was properly initialized before being used, especially ones being used in the line pointed to by the stack trace.
- Use printlines to determine **where** the error is occurring. Put them at the beginning and end of important methods, and see which ones print correctly.
- Once you find **where** the error is being thrown, use printlines to check the value of every variable at that point. If multiple variables are being used in a single line (ex. `_car.move(_grid.getLocation()) ;`), make sure to check the value of all of the variables.
- Remember that small things like a misspelled variable name or an out-of-scope local variable can lead to hard-to-spot errors.
- `ArithmeticException`: These occur through illegal arithmetic attempts. Most likely you tried to divide by zero.

**\*\*Tips to find/resolve bug\*\***

- Use printlines to check the value of variables you are performing operations with.

**\*\*Note:** Runtime errors may occur at any point while the program is running. For instance, if you have a game, you may only get a runtime error if you press a certain combination of keys, so make sure you test your programs thoroughly and in many situations!

## Conclusion

While this guide certainly doesn't cover every bug you will ever encounter nor all types of debugging methods, it is certainly a good start and should be quite helpful in this course. Remember that while your TAs are good resources for projects and assignments, their job is not to solely debug your code. As the semester goes on, you will be expected to debug as much of your own code as possible, and TAs have the right to turn you away from hours if they feel like you have not made sufficient efforts to find and resolve certain errors. Debugging is one of the most important skills to have as a programmer, so the earlier you begin to understand the different types of bugs and ways to fix them, the better off you'll be in the long run. And remember, you know your code better than anyone else, meaning you are the **most qualified** person to debug it!