## Overview

In this assignment, you’ll learn to download and set up starter code that we provide, a skill which you’ll need for several future programming assignment in this course. You’ll use that code to complete an implementation of the Ham and Spam counting game. Along the way, you’ll start to learn about Java [unit testing](https://en.wikipedia.org/wiki/Unit_testing), a repeatable, automatable way to detect bugs (and not incidentally, a way for us to test your submissions). Finally, you’ll learn to export and submit the project to the autograder.

## Goals

* Import a project into VSC.
* Execute code within VSC.
* Run tests within VSC.
* Write code within VSC.
* Export a project from VSC and submit it to the autograder.

## Some reminders

1. Start this (and future) assignment(s) early. It is your responsibility to start early enough that you can get help if you have trouble!
2. This is the first programming assignment. You may work a partner on this and all programming assignments.
3. Do not expect much, if any, partial credit if you submit a program that does not compile or contains an infinite loop. Errors of these sorts in your project will cause the autograder to fail, and you will receive a zero for your submission.
4. We do our best to check the assignments and autograder. But, if you think something is wrong with the autograder or assignment (as opposed to your submission), contact the course staff immediately. Capture a screenshot of the error, and export a copy of your assignment in the state that triggers the error. Providing these to us will help us immensely in narrowing down the problem.
5. Late submissions (past the self-granted extension, described on the syllabus) will not be accepted. Excuses with documentation may permit you to submit an assignment late on a case-by-case basis, consistent with the syllabus, the University’s [Academic Regulations](http://www.umass.edu/registrar/sites/default/files/academicregs.pdf) and otherwise at my discretion.

## Downloading and importing the starter code

Download and save the provided archive file containing the starter code – it’s compressed in the zip format. Note that in this and future assignments, the starter code is linked at the top of the assignment’s page.

Depending upon your browser and operating system, you may end up with the zipped archive file, or you may end up with the decompressed archive, which will appear as a folder (also known as directory), named hamspam-student. Put this folder into an appropriate location on your computer.

**A brief but important digression:** You should probably put your files for 210 all in one place. I strongly, strongly, strongly suggest you do one (or ideally both) of the following:

* Sign up for a cloud-based storage service, sync that storage with your local computer, and place your course directory inside this synced directory.
* As a member of the UMass community, you have access to Google Drive with basically unlimited storage space through [Apps at UMass](http://www.umass.edu/it/google-apps). You also have access to Onedrive through UMass, and many students have their own [DropBox](https://www.dropbox.com) accounts, and so on. Most of these services offer applications that perform file sync or backup or both.
* Enable automatic backups to an external drive, or sign up for a backup service. Make sure your 210 directory is being backed up.
* macOS has [Time Machine](https://en.wikipedia.org/wiki/Time_Machine_(OS_X)); Windows has [Windows Backup](https://en.wikipedia.org/wiki/Backup_and_Restore), and there are myriad third-party backup utility vendors. Find one and use it. Make sure you know how to recover from a backup *before* you need it.

In any event: remote storage is virtually free, and the files created for this course are tiny relative to, say, your carefully-curated K-pop MP3 collection. “My computer crashed” won’t be generally be acceptable as an excuse in this class.

**Now back to our regularly scheduled program.**

Open VSC. Depending upon if you’ve used it before, you may have various “Welcome to Visual Studio!” kinds of windows pop up. Read their contents if you like, and then close them. Eventually, you will have a single window left, that says something like “Visual Studio Code – Editing evolved” and a list of options. Choose the “Open folder…” option (or select it from the drop-down “File” menu), find your hamspam-student folder, and open it. **You must use “Open folder” and not “open file”** or VSC won’t know that the file(s) you are editing are part of a project and the tests won’t work.

You may be prompted to install the course-specific extensions if you haven’t already. (Do so unless you have reason not to.)

You may also see a prompt asking if you “want to exclude the Visual Studio Code Java project settings files […] from the file explorer.” I suggest you do so “in workspace” or “globally” – this will prevent you from accidentally breaking VSC’s Java configuration for this project.

You should see a hamspam-student folder in the Explorer window on the left.

Folder are represented by small arrows pointing down for folders whose contents is shown, and to the right for folders whose contents is hidden. Click on the folder name to toggle whether its contents is shown or not. In particular, click on src, support, and test.

You should see three Java source files among the folders: HamSpam.java, HamCommander.java, and HamSpamTest.java. You can open them by clicking on them. Do so, and take a look through each of them. You’ll see that they involve a game about “Ham”, “Spam”, and numbers, and that VSC has detected errors.

## src, support and test

In this (and all future) projects, you’ll see these three directories. What do they mean?

* src is where files you might modify will be located. You can do *almost* anything you want to these files – in particular, you can write new code and add new methods. But you *cannot* change the “signature” of methods that we’ve provided to you – you must keep their scope, names, parameters, and return types as written. (This is a limitation of the autograder and of Java – methods with different signatures are different, even if they have the same name!)
* support are files we’ve provided that you can use however you want – usually, they help “support” the code in src. *But!* The autograder will not see any changes you make in support! It always works from the original, unmodified version of these files.
* test contains “unit tests.” We’ll talk more about Java automated testing next week, but in short, these are short code snippets that check an “expected” (that is, correct) value is returned by your program’s code (called the “actual” value). Hopefully you’ve seen these already in Python in 190T/160! VSC and many other tools have the built-in tools to automatically run tests and tell you whether your code is “passing” them or not, and that’s also how the autograder works.

## On Ham and Spam

“Ham and Spam” is a children’s counting game. Before it is played, the players agree on a *ham number* and a *spam number*. Both are integers greater than one, and they cannot be the same number. The players then take turns saying the *hamspam* value for each successive integer, starting at one. Usually, the hamspam value is just the number. But:

* If the number they are supposed to say is divisible by the ham number and not the spam number they say “ham” instead of the number.
* If the number they are supposed to say is divisible by the spam number and not the ham number they say “spam” instead of the number.
* If the number they are supposed to say is divisible by both the spam number and the ham number they say “hamspam” instead of the number.

For example, if the ham number is three, and the spam number is four, then the first twelve hamspam values are:  
1, 2, ham, spam, 5, ham, 7, spam, ham, 10, 11, hamspam

For this assignment, you are going to modify the HamSpam class provided to produce the correct value or values for a game of “Ham and Spam”.

## Using VS Code

Let’s walk through using VSC to identify and fix errors, run code, and run unit tests.

**Finding and fixing compilation errors.** You can see that VSC has highlighted the HamSpam.java in red. The red underlines in the source for HamSpam.java indicate there’s an error; if you hover the mouse pointer over that line, you’ll see that “The blank final field hamNumber may not have been initialized.”

In other words, the instance variable hamNumber is declared (private final int hamNumber) but never set to a value if the class is instantiated. Your lazy instructor didn’t even provide you with code that compiles!

Fix the error by modifying the constructor of HamSpam to set the instance variable hamNumber to the appropriate argument of the constructor. (A reminder: when a local variable, such as the argument to the constructor, has the same name as an instance variable, you can disambiguate the instance variable by including the this. qualifier. In other words, you can write this.hamNumber = hamNumber; in the constructor to set the instance variable to the argument passed to the constructor.)

Then fix the same error for spamNumber.

Each distinct instance of HamSpam may have a different pair of ham and spam numbers. For example, an instances with a ham number of four and a spam number of five is created when a program calls new HamSpam(4, 5)). These values are tracked in these instance variables. When you open HamSpamTest.java you’ll see that more than one instance of HamSpam is created, each with its own ham and spam numbers.

**Running programs.** Once you’ve eliminated errors, you can run the program. But you’ll note there’s no main method in HamSpam.java. Instead, open HamCommander.java and take a look at its main method. To run it, right-click on HamCommander.java in the left-hand side of the window , then choose “Run” from the context menu. A “Terminal” showing the “Java Process Console” will appear in the bottom of the VSC window. Follow along, pressing Enter after each number you enter, to see the (sometimes incorrect) results of the current implementation of Ham and Spam.

If you want to check your implementation of HamSpam, you can do it this way. But now we’ll look at automated tests, which provide a faster way to more reliably repeat the same set of checks each time you run them.

**Running tests.** Have the file “HamSpamTest.java” visible in your editor, and then open the “Command Palette” using the “View” menu. (You can also use a keyboard shortcut to do so; on a Mac, for example, it’s Command-Shift-P.) You are looking for an item labeled “Java: Run all tests”. To find it quickly, type the word “tests” to narrow the listed choices. Choose the item and run it by clicking on it or pressing “Enter” when it’s selected.

Several things will now happen. VSC will run all the tests; some will pass, and some will fail. VSC will jump to one of the failing tests and show its results in the editor. A “TEST RESULTS” tab wil be added to the bottom pane, alongside whatever other tabs are there (typically some combination of “TERMINAL”, “PORTS”, “PROBLEMS”, and “OUTPUT”.) If you click “TEST RESULTS” you can navigate a list of the tests and their results.

You can also view tests more generally through the “Testing” pane, which you can find by clicking the “Testing” icon on the left side of the interface – it looks like a flask, and is usually below the “Extensions” icon, which looks like three connected squares and a fourth in the upper-right slightly removed.

When you do this, the left-hand side will witch to the testing interface. You should see a total of fourteen tests: four tests that pass, and ten tests that fail. If you don’t see them, click the small arrow to the left of “hamspam-student” to expand it – repeat this process with each revealed item until you see the list of tests.

Familiarize yourself with the testing interface: You can filter by test name, or you can click the funnel icon to limit the list to only failing tests; you can clicker the icon to the far right of each test to jump to the test’s source code (it looks like a piece of paper with an arrow pointing to it); you can click the name of each test to see the “stack trace” (that is, the currently-executing methods when the test failed).

For example, if you select testHamAndSpamGetValue, you should see the following under Failure Trace:

org.junit.ComparisonFailure: getValue returns incorrect value expected:[[hamspam]] but was:[[12]]  
 at hamspam.HamSpamTest.testHamAndSpamGetValue(HamSpamTest.java:47)```

…and much more of the stack track, but it’s typically only the top line or two that will be relevant.

JUnit tests work by checking that the expected result of a method call equals the actual result. The failed test indicates that one (or more) methods in the starter code aren’t returning correct values. Examine the test; again, click on the icon to its right, or navigate to it manually in HamSpamTest.java around line 46.

Here, you can see that the test case expects the result of hamspamThreeFour.getValue(12) to be the string "hamspam", but as the output in the Failure Trace indicates, it produced the string "12" instead.

**Diagnosing and fixing problems.** Go to the declaration of getValue(); you can do so by right-clicking on it in HamSpamTest.java and selecting “Go to Definition”, or by double-clicking on HamSpam.java in the Package Explorer and scrolling to it.

Your goal is to correct the implementation of the getValue() method. The correct solution is not merely to make the function return “hamspam” when n == 12. Instead, you should revise the code so that the function will return the correct string for any n.

There is at least one thing to fix. The getValue() method currently checks for equality against numbers, not divisibility. You can use the modulus (remainder) operator (%) to check for divisibility. It divides one number by another and returns the remainder. For example, if you wanted to print out whether the value of a variable n was divisible by three, you could write:

if ((n % 3) == 0) {  
 System.out.println(n + " is divisible by 3");  
else {  
 System.out.println(n + " is not divisible by 3");  
}

Notably, the value ((n % 3) == 0) will evaluate to true if and only if n is divisible by three. You can use this fact in your implementation of getValue(). One straightforward correct implementation will involves three if statements. Post a question on CampusWire or come chat with us during office hours if you get stuck.

**Finishing up.** Next, you’ll need to write getValues(), which will allocate and return an array of String containing the correct hamspam values for the range. Think carefully about what you need to do: create and return an array of the right size with the right values. Break this down into several parts: How big should the array be? Then, how do you put the right values into the right places of the array?

When we autograde your program, we will test both getValue() and getValues() with other ham and spam numbers to make sure each works properly. Our test cases will obey the constraints described in this assignment (for example, the ham number and spam number will always be greater than one, and never be equal to one another), but are otherwise unconstrained.

In this assignment, we’ve given you all the test cases. But in future assignments, there are test cases we are going to use that you won’t be able to examine for yourself. You will have to think about the problem and make sure you’re not missing any details or corner cases if you want to pass them. (We’ll teach you to write tests soon, but there’s no magic: copy/paste and edit an existing test to get 90% of the way there – it’s the same as in Python.)

You can also use the HamCommander class for interactive testing. But as you’ll see, just being able to “Run tests” quickly and repeatedly is much more convenient, especially as programs grow more complex.

## What not to do

(I apologize for the tone of the following, but I need to be absolutely clear about something that is not permissible.)

This is the first assignment where you may be tempted to game the autograder, as many of the static methods are standalone “functions,” with well-defined behavior that depends only upon their input. Don’t do it.

What do I mean by gaming the autograder? Since we are still giving you the tests, you may be tempted to write code that is not general, but that does pass the tests. Put another way, you may be tempted to hard-code the “correct” answers to the tests (that won’t work on other inputs!) into your methods via a series of if/then/else statements that work on very specific values (like ham=2, spam=3) but fail for others.

**Do not take this approach.** We expect you to write *general* solutions to the assignments, not just attempt to pass the tests. We give you the tests to help you learn (though note that later in this class, and in life generally, you won’t always get the tests handed to you), not to let you pattern-match if-then-else statements on the test cases.

Doing so is gaming the autograder. We will treat it as academic dishonesty and lobby for the maximum possible sanction – at least an F in this course, if not an academic suspension.

If you are having trouble understanding the assignment or need help, please ask! But don’t game the autograder.

## Another things to perhaps not do

You might also be tempted to use Copilot to solve this assignment. To be frank: Copilot (and ChatGPT, etc.) eat introductory CS assignments, especially ones with well-written comments (like this one), *especially* ones where a solution is probably already on GitHub somewhere (like this one), for breakfast.

Should you use Copilot or the like to solve this assignment? You could. I’m not going to formally consider it cheating. But you are responsible for knowing the things you need to know to complete this assignment or things like it (on, say, quizzes or exams), including Java syntax and commonly-used methods and classes from the Java standard library. If Copilot helps you get there, great. But if you use it as a magic genie to solve the assignment, do keep in mind it’s not going to be there for you on the quizzes or exams, so you’ll want to *understand* what it’s doing. And if it makes subtle mistakes in logic or semantics – which it probably won’t on simple assignments like this one, but may later – that’ll be on you to figure out.

## Submitting the assignment

When you have completed the changes to your code, you should export an archive file containing the entire Java project. To do this, open the Command Palette again, and choose “Archive Folder”. This will create a zip file containing the project in the same folder where you stored the hamspam-student folder. It will have a numerical suffix, for example, hamspam-student\_1598196069605.zip. (You can also make the zip file yourself with another tool, if you prefer.)

Next, log into Gradescope, select this assignment, and submit the file for grading. You’ll see the contents of the .zip file in the upload window after you select it.

If you are doing a submission with a partner, *be sure to note that when submitting!* Make sure you select your partner in the submission dialog box on Gradescope. Only one partner needs to submit the file; both will receive the same credit.

Compilation errors will be provided by the autograder, but the exact details of the tests we run on your code might not be provided in Gradescope. For this assignment, the tests you have locally and the tests run on Gradescope are the same, but this won’t always be the case. You must read assignments carefully – if your submission is not passing a test, it is almost certainly because your submission doesn’t match the requirements of the assignment.

Remember, you can resubmit the assignment as many times as you want, until the deadline. (Make sure you upload the right version!) If it turns out you missed something and your code doesn’t pass 100% of the tests, you can keep working until it does.