Lab 2: Intro to Debugging and UMLet

Debugging

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

In the design phase, you decide which features you need to implement and how to organize the classes in order to do so. 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 and being able to debug them is an essential skill.

Goal: Gain familiarity with different types of bugs that are encountered while programming and learn elementary techniques for resolving them.

NOTE: This is an extremely important skill, not only for CS15 but for any and all of your classes. Read this lab carefully, we promise it will end up saving you lots of time and grief!

Incremental Coding

Bad Coding: Type all the code, try to compile it, get a hundred errors, run to a TA for help because you have no idea what's going on anymore.

Good (Incremental) Coding: Fill in a method and try to compile it. If it compiles, great! Now call the method. Didn't get the expected results? Look through the method, fix the bug and try again! Worked? Awesome! Fill in the next method. Rinse & repeat.

Good programmers will have bugs too. Incremental coding just ensures that when you run into a bug, you’ll have a much better idea of where it is. Even if you can’t figure out the bug, it will make it much easier for TAs to help you.
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 of the parentheses.

String Boot Camp

A String is a built-in Java object that represents text. There are a couple ways to instantiate a string. You can create a variable to hold a string...

```java
String myString = "CS15 TA";
```

...or you can just create a string when passing it in as a parameter:

```java
System.out.println("CS15 TA");
```

The quotes around CS15 TA indicate that it is a string.

Hello, World

- Install the lab: “cs015_install lab2”
- Navigate to the ~/course/cs015/lab2/strings directory from the terminal and enter the command “atom Printer.java &”. This will open up your Printer.java file in Atom.
- In your terminal (you should still be in strings), type “javac *.java”. This will compile all .java files in that folder. There should be no errors, so nothing should print to the terminal.
- To run the code, type “cd ..” to move back up to the lab2 directory, and then type “java strings.Printer” into the terminal to run the code. Again, nothing should print.
- Go back to the Printer.java file. In the constructor (the method called Printer), add a printline so that the program will say “Hello, World!” when it runs. Compile your program in strings and run it in lab2.

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 crazy), 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.

```java
String myString = "My name is" + " CS15 TA" + "!";
System.out.println(myString);
```

Output:
>> My name is CS15 TA!

The previous example may seem a little pointless. The real power of concatenation comes when you use it like this:

```java
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).

**Two Notes About Printline Style**

1. Please make sure to **remove all printlines before handing in a project**. It is bad practice to leave printlines 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 printlines have context. For example, don’t print

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

instead, print

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

---

**Hello, World x 3**

- Go back to the Printer.java file
● Add two more unique ways to print “Hello, World!” to the console.
  ○ At least one “Hello, World!” should use a local variable that you create.

● Return to the terminal and compile your program. Debug if necessary (make sure your syntax matches our examples!).

● Run the program. It should print out “Hello, World!” to the terminal three times.
  ○ Note: The main class of this program is Printer. After compiling, the program can be run with “java strings.Printer” from the lab2 directory

---------------------

Make sure to save this program so that you can show it to a TA at the next checkpoint! You should have both the terminal output and the code ready to show.

If you want to learn more
You can take a look at all the other things you can do with Strings by reading the Javadocs.

Commenting out Code

Another useful debugging tool is commenting out code. You’ve already seen comments explaining 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, 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 Grinch class method stealChristmas(...). However, we haven’t yet defined the method stealFood(). We can comment out the lines that use that method in order to test the other lines:

```
1 public void stealChristmas() {
2     this.stealTree();
3     this.stealPresents();
4        // this.stealFood();
5     System.out.println("Christmas stolen? " + this.isHeartGrowing());
6 }
```

If we ran this code without disabling line 4, we’d get an error since our method stealFood() hasn’t been defined yet. Commenting out these lines lets us code stealChristmas(...)
incrementally, making sure that `stealTree()` and `stealPresents()` work before moving on to `stealFood()`.

**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 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!

**Runtime** bugs indicate an error in the logic of the code. 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 shows up in another). As time goes on, most of your time will be spent on runtime bugs, some of which can be extremely tricky.

**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:

```
Class name: App.java
Line: 18
Message: ';' expected
```

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 of the common ones.
  - o error:
    - ■ ‘;’ expected
    - ■ ‘)’ expected
    - ■ <identifier> expected
  - o Tips to find the 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!** Most likely you forgot a bracket or parenthesis.
    - ■ Note that the compiler's suggestion for what should be fixed is not always correct! Sometimes it will say "';' expected " when actually you forgot a "=" , for example.

- Method or Variable Errors: These are generally caused by using a method or variable name which is incorrect
  - o error:
    - ■ cannot find symbol
  - o Tips to find the 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.

If you feel unsure of Java syntax, you should spend a little time reviewing it after this lab. **Make sure you can declare instance and local variables, and define and call methods.** It will save you a lot of time and frustration in the long run. Look back over the slides or go to TA hours if you have questions!

Getting your program to compile

We’ve created a program to tell you what is at the Ratty today. The only problem is that it is very buggy.

- In Atom, go to File -> Open Folder and then navigate to the folder named ratty.
- In your terminal, navigate to the ratty folder.
- In your terminal, type “javac *.java”. This will compile all .java files in that folder.
• The terminal will print out a number of errors like the ones above. Look at RattyMenu.java, and start trying to fix the syntax errors.
  ○ Look over the file and fix any syntax errors. When you can’t find any more, save the file and then type “javac *.java” into the terminal again. If there are more errors, go back to RattyMenu.java and try to find them.
  ○ When you type “javac *.java” into the terminal and the compiler throws no more errors (it doesn’t print anything), you’re done!

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:

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 and 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 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**:
  - **Cause:**
    - 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.
    - E.g.: The following code will result in a NullPointerException.
      ```java
      String name; // name hasn’t been initialized
      System.out.println("Hello, " + name);
      ```
  - **Tips for finding the 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.
      - Remember, an *instance variable* needs to be declared and initialized separately! Look at lecture slides if you’re not sure of the syntax for this.
    - Use printlines to determine where the error is occurring
      - Put printlines 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 (i.e. `_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**:
  - **Cause:**
    - Illegal arithmetic attempt. Most likely you tried to divide by zero.
  - **Tips for finding the bug:**
    - Use printlines to check the value of variables you are dividing by.
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!

**Hand Simulation**

This is a technique for debugging which focuses on looking carefully at your program and trying to find and fix bugs before they occur. There is nothing more frustrating than running your program over and over and trying to change little things until it works. It will take longer, and you often won’t understand what caused the error even if you solve it. *Instead we recommend that you walk through all of the logic in your program before you run it, following the flow of control of the program from start to finish.*

1. Check that variable names are correct.
2. Check that you are calling the correct methods at the correct places with the correct parameters.
3. Use a piece of paper to keep track of the value of each variable at each stage in your program.

Doing this will be faster, and ensure that you understand how your program runs. Later on, you’ll need to use this technique to find complex bugs that Java won’t pick up on.

**Trying Hand Simulation**

- Use hand simulation to understand what happens when a `LemonadeStand` is instantiated:

```java
public class LemonadeStand {
    private int _profit;
    private int _hoursWorked;

    public LemonadeStand() {
        _profit = 0;
        _hoursWorked = 0;
        this.runLemonadeStand();
    }

    public Lemonade makeLemonade() {
        // Code to make lemonade
        return lemonade;
    }

    public Stand setUpStand() {
        // Code to make a stand
```

```java
```
public void sellLemonade(Lemonade lemonade, Stand stand) {
    // Code to sell lemonade to 20 customers using a stand
    _profit += 20;
}

public void runLemonadeStand() {
    Lemonade lemonade = this.makeLemonade();
    Stand stand = this.setUpStand();
    this.sellLemonade(lemonade, stand);
}

public int getHourlyProfit() {
    return _profit / _hoursWorked;
}

Hint: “+= 20” increases the value of the variable by 20.

- You should have a good understanding of what happens when a LemonadeStand is instantiated and what the final values for each of the instance variables are. Using this information, discuss with a neighbor what would happen if getHourlyProfit() were called.

Checkpoint 1: Call over a TA and explain what happens when a LemonadeStand is instantiated! You should also be able to explain what happens when getHourlyProfit() is called. If you did work, keep that open to show the TA as well! Also have the TA check your Hello World program.

We’ve now given you a number of different tools to debug runtime errors. Using these techniques (hand simulation, print lines, and terminal exceptions), let’s debug the following program:

Getting Your Program to Run

- Go back to your ratty code.
- Debug the program until it runs as intended.
- The following should print to your console when the program runs correctly (you will have to add the last line by yourself):
  >> Here is today’s Ratty Menu.
  >> We are expecting <a random integer between 60 and 260> people for dinner.
  >> There is Greek Salad and Beef Chili for dinner.
Each dish of Chocolate Cake only feeds 20 people, so we have to make enough dishes for everyone to eat chocolate cake dishes.

Sincerely, Chef <your name>.

Note: To run your program, type “java ratty.App” from the lab2 directory.

Also make sure you continue to save and compile your program as you make changes or they will not be reflected. You can compile from inside the ratty directory using “javac *.java” or from the lab2 directory using “javac ratty/*.java”.

Checkpoint 2: Once your program runs and prints the above message, call over a TA. Make sure to have both your terminal and Atom window open.

UMLet and Program Design

Now that we know how to use classes and create Java programs, it's time to introduce you to another important aspect of computer programming: design. Up to this point, our projects have been small and you may not yet realize the benefits of good design, but trust us, the time will come. Well-thought-out designs will benefit you as your projects increase in scope. These designs can be drawn out in diagrams called Unified Modeling Language (UML) diagrams.

This lab will teach you the basics of design, UML diagrams, and inheritance trees. It will also introduce you to UMLet, a program designed to make your UML diagrams easy to read and easy to make.

Goal: Learn to create UML containment and inheritance diagrams in UMLet!

Introduction

Let's begin by familiarizing ourselves with UMLet. UMLet is a very simple diagram editor that will help you make readable and easily editable UML and inheritance diagrams. You will use it in your answers to design discussion mini-assignments. While design diagrams for LiteBrite and TASafeHouse are not too complex, later programs will have numerous classes and associations, all of which can lead to illegible design diagrams when done by hand.

Now, let us review the features of UMLet that will be most useful to you.
Getting Started

- Start UMLet by typing “umlet &” into a shell.

Remember: Appending “&” to the end of a command keeps your terminal free for additional commands and programs.

The UMLet Interface

1. **Editor region:** This is where your active diagram can be found. The commands for the editor are your standard Windows shortcuts (i.e., CTRL + S to save, CTRL + C to copy).
If more than one file is open, they will appear in the form of several tabs at the top of the editor screen for easy access and viewing. Components in this window can be dragged around, enlarged, rotated, and otherwise manipulated with the mouse.

2. **Components region:** This is where you can find components to add to your diagram. Double click on a component in this region to add it to your diagram.

3. **Help/Text Editing Region:** When an element in the Editor Region is selected, this area turns into a text editing region for the element. For example, to change the class name of a block, type it in this area.

**Types of Diagrams**

There are two types of diagrams we’ll be covering in this lab: containment diagrams, which show where each class in a program is instantiated, and inheritance diagrams, which show how child classes extend their parent classes.

**Containment Diagrams**

Containment diagrams are graphical representations of class relationships within a program.

This is a containment diagram:

![Containment Diagram](image)

Here’s how we would interpret the diagram above.

Consider the classes **Andy, CS15Lecture, CS15TA, and CS15Student**—all of these classes are concrete so they can be denoted by a box with their non-italicized name. One instance of Andy is contained in the class **CS15Lecture**, so the containment is denoted by an unfilled
diamond arrow. There are several instances of CS15TA and CS15Student contained in the CS15Lecture class: these containments are denoted by filled diamond arrows.

There is an instance of TabletComputer (a concrete class) in class CS15Lecture as well. Andy needs to know about his TabletComputer. Therefore, Andy has a reference arrow to the instance of TabletComputer in CS15Lecture.

Here are more formal definitions for all of the components:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Name</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>SimpleClass</td>
<td>Concrete Class Box</td>
<td>Classes are represented by a box with its name. “Concrete classes” are classes that can be instantiated.</td>
</tr>
<tr>
<td>AbstractClass</td>
<td>Abstract Class Box</td>
<td>Abstract classes are represented by a box with its name in italics.</td>
</tr>
<tr>
<td>ClassB</td>
<td>Unfilled Diamond Arrow</td>
<td>Shows that a single instance of a given class (ClassA) exists in the class the arrow points to (ClassB)</td>
</tr>
</tbody>
</table>
Filled Diamond Arrow

Shows that multiple instances of a given class (ClassA) exist in the class the arrow points to (ClassB).

Reference Arrow

Indicates that a class (ClassA) has a reference to — “knows about” — the class to which the arrow is pointing (ClassB).

Woohoo! Now that we know the lingo, let's try creating one of our own!

Create your own Containment Diagram

- Your containment diagram should include:
  - An App class (just like the diagram we gave you for LiteBrite).
  - The Aquarium
  - An Aquarist (like a zookeeper, but for an aquarium)
  - Fish
    - Do not use a specific type of fish; let Fish be a general class.
  - Fish Tanks
  - Visitors

Hint: The App class should contain the top-level class, the Aquarium.
Consider: Should the aquarist know about the fish?

Make sure to save this diagram, you will need to show it to a TA at the next checkpoint!

Congratulations! You've just completed your first UML diagram!
Inheritance Diagrams

Inheritance diagrams show the relationship between a superclass and its subclasses. Inheritance diagrams, or trees, look a lot like family trees, and rightfully so. Much like you inherited your eyes and hair from your parents, a child class inherits the traits of its parent. This means that the subclass (or child class) inherits all of the methods of its superclass (or parent class).

Remember from our UML containment example that CS15Lecture contained both CS15Student and CS15TA. CS15Student and CS15TA are different classes, but they have many similarities. If we were to write both classes from scratch, we would have to write a lot of code twice. To be more efficient, we will write a superclass BrownStudent that both subclasses will inherit from.

Interfaces are also included in inheritance diagrams. Interfaces are similar to classes, except that they only define what capabilities classes must have, but not how classes will implement them. Classes can implement multiple interfaces. Any subclass also inherits the interfaces of its superclass. A subclass must implement any interfaces of its superclass.

We see that the class BrownStudent implements the interface Teachable (represented by a circle). Because both CS15TA and CS15Student inherit from BrownStudent, both CS15TA and CS15Student are also teachable! The Teachable interface could have methods like CompleteAssignment and ReviewMaterial. CS15TA and CS15Student would then both be able to complete assignments and review material.

Note: As in the example above, inheritance diagrams should only use vertical or horizontal lines.

Inheritance Diagrams: Symbols and Meanings

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Name</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Inheritance Arrow

ClassA extends or inherits from the ClassB.

Interface Circle

ClassA implements InterfaceB

Remember: Inheritance diagrams and containment diagrams are separate. An inheritance diagram should hold information about interfaces and superclasses, whereas a containment diagram models the implementation of the code. To make this clear, you should put the diagrams in separate files.

Inheritance Diagrams

- Open a new file for your Inheritance diagram (File >> New or Ctrl + N)
- Let’s make our aquarium a little more interesting: Design an inheritance tree that allows for many different fish.
  - Include at least 3 species of fish.
- Then find at least two other classes in this example that could share a superclass. Think of a superclass and draw an inheritance diagram for this relationship as well.
  - Add an interface which these classes could both implement
  
  Hint: These two classes could also share a parent class with you and me!

Make sure to save this diagram, you will need to show it to a TA at the next checkpoint!

More Complex UML Diagrams

Suppose that we want a method in the top-level class called “makeAllStudentsAttentive” such that all students will look at Andy. This would require all of the students’ eyes to be on Andy. All BrownStudents should have Eyes; this trait is not unique to CS15Student. So, the superclass BrownStudent should contain the two instances of the Eye class. Because CS15Student is a subclass of BrownStudent, it will also inherit two Eyes. However, since
the Eyes are created in the BrownStudent superclass, that is where we draw our Eyes in the UML diagram. Therefore, we would show this more complex containment relationship as below:

<table>
<thead>
<tr>
<th>Containment Diagram</th>
<th>Inheritance Diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Containment Diagram" /></td>
<td><img src="image" alt="Inheritance Diagram" /></td>
</tr>
</tbody>
</table>

We display BrownStudent as containing multiple instances of Eye in the same diagram as the CS15Lecture even though there are no connections between the actual class of BrownStudent and the CS15Lecture itself. However, we know that CS15Student extends ("is a") BrownStudent, which means that CS15Student will contain Eye instances too.

This is an important concept for TASafeHouse and the rest of the course, so don’t hesitate to ask a TA for further explanation/clarification if you are confused!

### Conceptual Wrap-up

- In a text editor (type “atom &” into your terminal), answer the following two questions:
  - What is the difference between an Inheritance Diagram and a Containment Diagram? (Please explain fully)
  - What is the relationship between Eye and CS15Student?

### Checkpoint 3:
Show your diagrams and answers to a TA to be checked off. Be sure you can explain them all thoroughly. Congratulations on finishing the lab!
UMLet in the Future

Now you can design and create diagrams with the best of them!

Remember to use UMLet for all of your design discussion mini-assignments when diagrams are required. Not using UMLet will result in deducted points.

UMLet is available for your personal computer, if you don't feel like schlepping to the Sunlab for your diagrams. The download is available from umlet.com (use the stand-alone version). If you have problems, please ask the Sunlab Consultant.