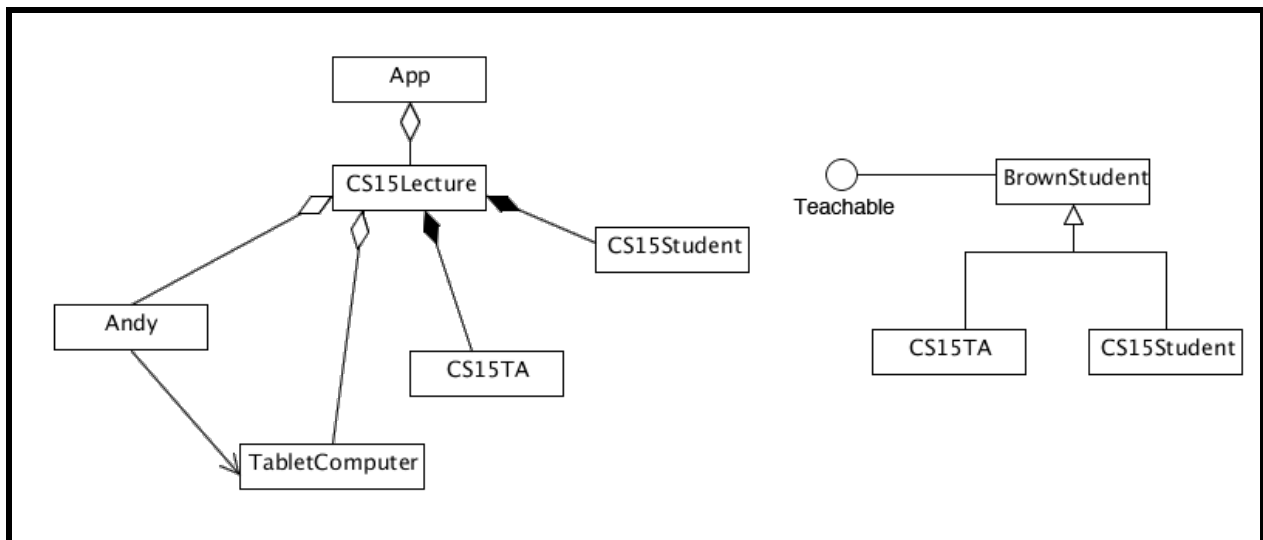


Introduction to UMLet and Design

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 (cough - Tetris - cough). 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. By the end of this lab, you will have the skills to create diagrams as coherent and clean as this one:



Goal: Learn to use UML diagrams, inheritance trees, and UMLet!

Introduction

Let's begin by familiarizing ourselves with UMLet, the program on which you will create all of the future diagrams required by design questions. UMLet is a very simple diagram editor that will help you make readable and easily editable UML and inheritance diagrams. 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. UML diagrams done by hand are also difficult to edit. UMLet can eliminate this hassle if used properly.

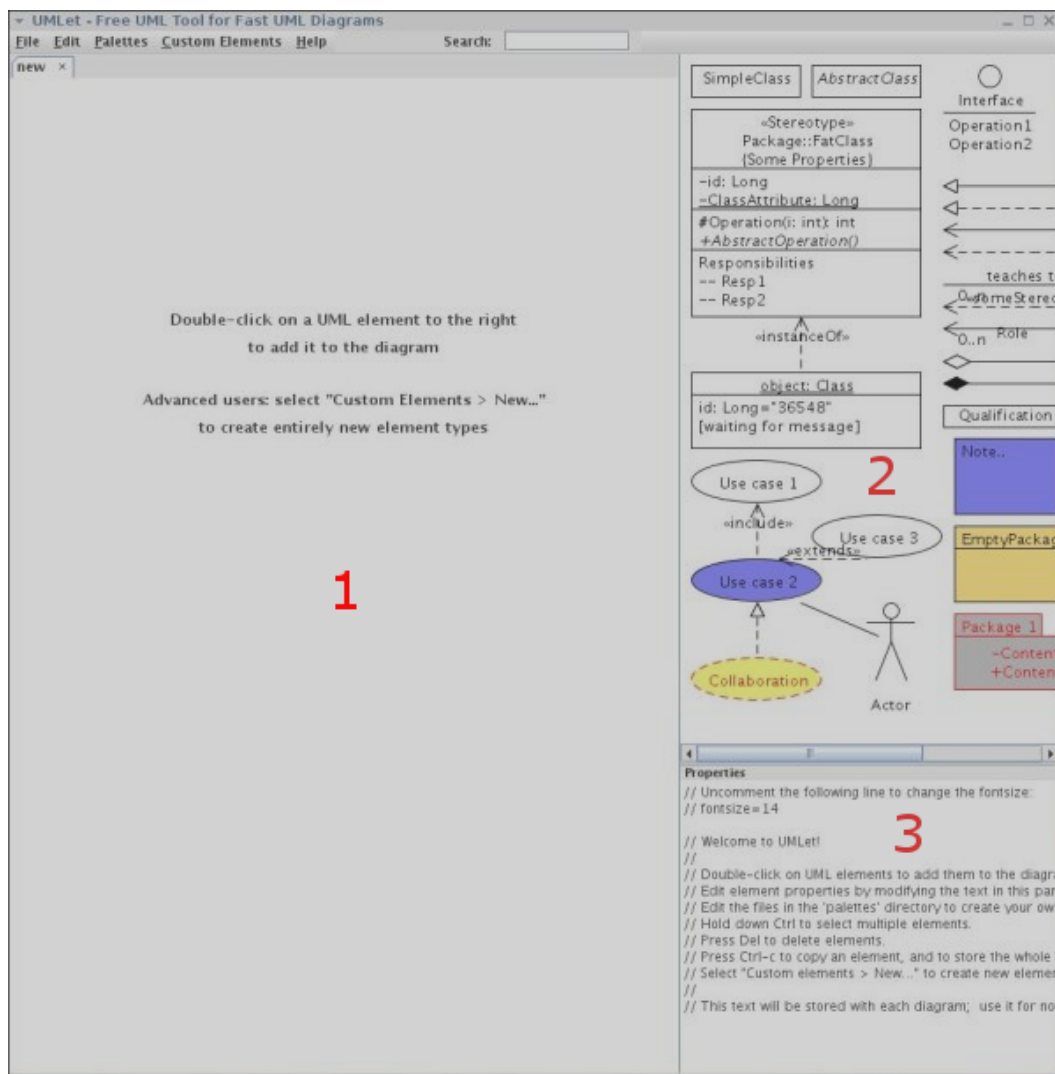
Now, let us review the features of UMLet that will be most useful to you.

Check Point 1: Getting Started

- Start UMLet by typing 'umlet' into a shell.

Remember: Appending '&' to the end of a command to keep your terminal free for additional commands and programs.

The UMLet Interface

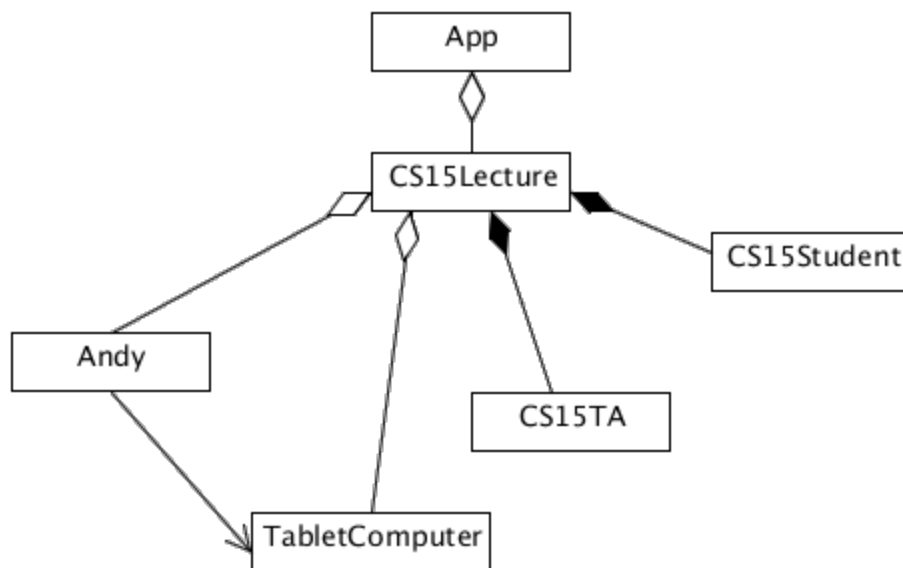


1. **Editor region:** This is where your active diagram can be found. The commands for the editor are your standard Windows shortcuts (i.e., CTRL-s to save, CTRL-c to copy). If more than one file is open, they will appear in the form of several tabs at the top of the editor screen for easy access and viewing. Components in this window can be dragged around, enlarged, rotated, and otherwise manipulated with the mouse.
2. **Components region:** This is where all of the components of your diagrams reside. To add a component to your editor region, double click on the component. And don't worry, you will *not* need all of these symbols for your diagrams--we will go over which symbols you do need to know.
3. **Help/Text Editing Region:** When UMLet is first opened, this box has all of the essential elements of UMLet explained, in case you should forget how to do something. You can also change the text size by following their instructions. When an element in the Editor Region is clicked, this area turns into a text editing region for the element (for example, to change the class name, type it in this area). When you click on the blank canvas, this area will turn back into a help area.


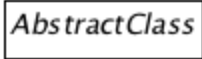



Containment Diagrams

Containment diagrams are graphical representations of class relationships within a program. Now that you're familiar with UMLet, let's review the essential elements of containment diagrams that you will be using.

This is a containment diagram:



Containment Diagrams: Symbols and Meanings

Symbol	Name	Use
	Concrete Class Box	Classes are represented by a box with its name. “Concrete classes” are classes that can be instantiated.
	Abstract Class Box	Abstract classes are represented by a box with its name in italics.
	Unfilled Containment Diamond	Shows that a single instance of a given class exists in the class the arrow points to.
	Filled Containment Diamond	The filled diamond at the head of the arrow indicates that multiple instances of a given class are contained in the class that the arrow points to.
	Reference Arrow	Indicates that a class has a reference to—“knows about”—the class to which the arrow is pointing.

For clarification, here is a written explanation of how these symbols might be used:

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 would be denoted by an unfilled diamond arrow. There are several instances of CS15TA and several instances of CS15Student contained in the CS15Lecture class; those containments would be denoted by the filled diamond arrow.

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.

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

Check Point 2: Containment Diagram

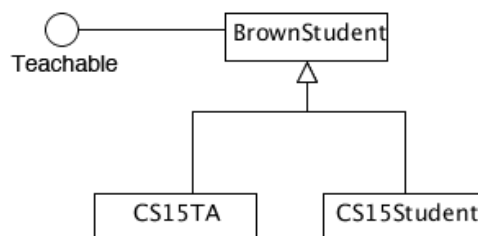
- With a partner, discuss the following containment diagram.
 - **NOTE:** The CS15 collaboration policy dictates that you can't take away any notes from your design discussions, and you cannot draw diagrams together--talk about design with your partner, but create your own diagram.
 - Your containment 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?
- **Show and explain your containment diagram to a TA before you continue.**

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. As you've just learned, a child class inherits the traits of its parent, much like you inherited your eyes and hair from your parents. However, unlike you, Java classes can only have one parent.

To demonstrate an inheritance diagram, let's revisit our earlier UML example. As you remember, in CS15Lecture, we had both CS15TAs and CS15Students. Even though these classes are different, they have many similarities. So, if we were to code this, it wouldn't make sense to write a CS15Student class and write a separate CS15TA class from scratch because there would be a lot of code that could be factored out. Therefore, a superclass, such as "BrownStudent," would be a good idea! This inheritance structure would be diagrammed like this.



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, such as which class instantiates another or what class needs to reference a different class. It is a good idea for your inheritance diagram to resemble the example and work itself down vertically.

Inheritance Diagrams: Symbols and Meanings

Symbol	Name	Use
	Inheritance Arrow	This means that this class extends or inherits from the class to which the arrow is pointing.
	Interface Circle	A class connected to the circle (which represents the interface) implements the given interface.

The above inheritance diagram explained:

We see that the class BrownStudent implements the interface Teachable (represented by a circle). Because both CS15TAs and CS15Students are types of BrownStudents, they should extend the BrownStudent class (represented by the inheritance arrows).

Note: It's perfectly fine to combine the two inheritance trees (1. BrownStudent implementing Teachable and 2. CS15TAs and CS15Students extending BrownStudent) because by nature of inheritance, the two subclasses implement the same interface, which creates a relationship between these classes!

Check Point 3: Inheritance Diagrams

- Let's make our aquarium at least kind of interesting: Design an inheritance tree that allows for many different fish.
 - Include at least 3 species.

Consider: This is an inheritance diagram. Should it be connected to your containment diagram?

- Find two other classes that could be generalized into another class. Draw an inheritance diagram for these classes as well.

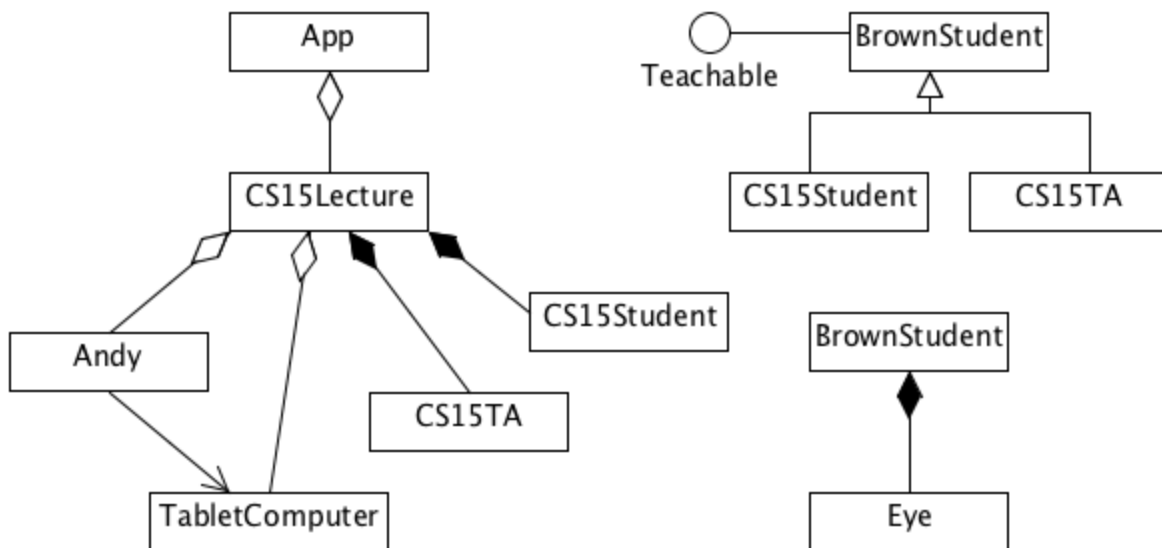
Hint: These two classes could also share a parent class with you and me!

- **Show a TA your inheritance diagram.**

More Complex UML Diagrams

The containment and inheritance relationships between classes can become tricky to display.

Consider that we want a method in the top-level class called “makeAllStudentsAttent” such that all students will look at Andy. This would require all of the students’ eyes to be on Andy. We don’t want to show CS15Student as containing the two Eye instances in the above UML diagram—that is not the true containment relationship! 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 contain two Eyes. Therefore, we would show this more complex containment relationship as shown below:



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!

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

**Remember to use UMLet for all of your design questions 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). Be

aware, in order to run this version of UMLet, you must have the latest version of Java, “Java SE 7”. If you run into problems trying to run UMLet on your own computer and have questions, please contact the Sunlab Consultants.

Collaboration and Design

Collaboration and design go together like cake and icing. Like peanut butter and jelly. Like programming and caffeine. There will be times when the program specs seem impossible, or when you have no idea where to begin, and talking through the design of a program could be just the kick-start you need!

CS15 allows collaboration on design after the Cartoon project, but only under certain stipulations (see the Collaboration Contract, available on the website, for the conditions for acceptable collaboration). Design allows plenty of opportunity to work together. We encourage you to discuss design with your classmates or with us! Just remember to not take notes away from your discussion (use a whiteboard and erase it!) and to let us know who you discussed your design with in your header comments.

Check Point 4: Wrap-up

- In a text editor (type `sublime &` into your terminal) , explain the difference between an Inheritance Diagram and a Containment Diagram.
- **Keep your answer open to show to a TA when you get checked off at the end of the lab.**

Debugging (Part 1)

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 lab, not only for CS15 but any and all of your classes.

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.

Having bugs in your code does not indicate that you are a bad programmer. In fact, even the best programmers can't write bug-less code on the first try. The trick is to identify those bugs in the beginning stages of your program. After all, it's a lot easier to find a missing parenthesis in 10 lines of code than trying to fix 45 bugs in 1000 lines of code. This is where incremental coding comes in.

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, but it didn't compile? Well, the first method compiled, so it must be syntactically correct. That means the problem is in the last method you filled in, so look at that the code, fix the bug, and compile again!

As you can probably guess by now, incremental coding is the practice of testing small portions of your code as you are writing them. It is MUCH easier to find bugs this way, since you will have less code to work with at any one time. You will also become much more familiar with your code (know what works and what doesn't), and the TAs will be much happier helping you tackle hard programming decisions rather than spending all their time scanning through pages of code trying to find the missing semi-colons and undefined variables.

Let's see how incremental coding could have worked for AndyBot:

1. Make AndyBot andy take a single step.
2. *Test: Make sure that the maze appears and that andy steps forward.*
3. Next, see if andy can successfully walk through the first tricky part of the maze.
4. *Test: Confirm that andy isn't hitting any walls.*
5. Slowly add more code until andy can make it all the way to the roadblock.
6. *Test: After adding a couple lines of code, make sure the program is still working as intended--it's easier to add more lines than to edit previous ones.*
7. Fill in the the solveRoadBlock(int x) method and call it.
8. *Test: Make sure the password is successfully submitted.*
9. Add in the final steps to leave the maze.
10. *Test: Run your code to see if you get the "Winner!" message.*

Types of Bugs

All bugs can be broken down into two categories: compile-time bugs and run-time bugs.

Compile-time bugs happen when you violate the rules of the programming language and the compiler has no idea 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.

Eclipse compiles your code as you type and will underline any compile-time errors with a red squiggly line (like a typo in MS Word). This is an exceptionally helpful feature that allows you to catch bugs as you type. This feature will save you a ton of time! Appreciate Eclipse!!

Run-time 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 you become more and more familiar with Java, compile-time errors will become easy to fix and much more of your time will be spent on run-time bugs, some of which can be extremely tricky.

Resource: Online TA

Go to http://www.cs.brown.edu/courses/cs015/compiler_errors.html for specific descriptions of different types of errors that you can get and how you can fix them. You can find this link off of the Online TA tab on our website.

Use that link BEFORE signing up for TA hours.

Most of the time you can fix a compile-time bug in *5 minutes* instead of waiting *half an hour* for a TA to find where you missed the semi-colon.

NOTE: Don't forget that Java is case sensitive. If you define a method `getColor()` and call `getcolor()` in your program, Java will look for the undefined method `getcolor()`. The Java compiler will then give you an error.

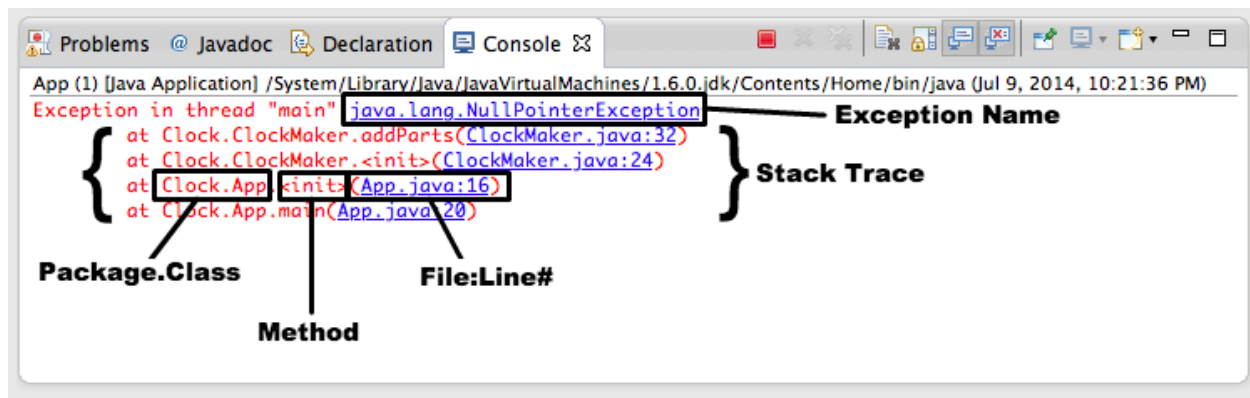
Run Time Errors

For runtime errors, things can get a lot trickier.

If your program crashes, Java will print the **stack trace** to your terminal. The stack trace is a list of all of the methods that were being executed when the program crashed. You'll find that the bug is almost always in one of those methods.

The topmost method is the method that got called most recently. Most likely, it's where the error happened. Moving down from there, you can trace the sequence of methods that were called to the first method that was called.

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 CS015 support code methods) and your methods are usually near the top. *You can safely assume that the Java base code and the CS015 support code is bug free*, so you should begin tackling the stack trace when it first enters one of your methods.



1. **Stack Trace:** The list of all the methods executing when the error occurred. Start from the top and look for the first method that you wrote, as the error probably occurred there.
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.

Common Runtime Errors

NullPointerException:

Thrown 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, or something that no longer exists because it was garbage collected or fundamentally altered.

Null pointers will occur most often when you create an instance variable and try to use it before you've actually assigned it a value. See below for an example of this occurrence.

```
public class OperateCar(){
    private Car _car;

    public OperateCar(){ }

    public void drive(){
        _car.moveForward(1);
        //NullPointerException occurs
    }
}
```

The above code would be fixed by initializing `_car` in the constructor, which would look like this: `_car = new Car();`

ClassCastException:

Happens when your code attempts to wrongly cast an object to a particular class. Specifically, your object is not an instance of this class, or your object is not a subclass of this class. For example, if you try to cast an `Car` to a `Color`

```
Car myCar = new Car();
Color myColor = (Color) myCar; //generates
ClassCastException
```

Arithmetic Exception:

Illegal arithmetic attempt. Most likely you tried to divide by zero.

StackOverflowException:

More methods got called than Java memory can handle. It is most likely you have an infinite loop or a problem with your base case in recursion, which you'll learn about a little later.

Note: Sometimes it may take a while before the `StackOverflowException` is actually thrown. If your program is taking a really long time to run, then you *might* be in the middle of an infinite loop.

```
Boolean alwaysTrue = true;
while(alwaysTrue){           //will run infinitely causing an
    error
    myCar.drive();
}
```

IndexOutOfBoundsException:

When Java tries to access an Object in a list at an index that is not in the list's range.

```
Car[] listOfCars = new Car[5]; //array with indexes 0
through 4
for(int i = 0; i <= 5; i++){
    listOfCars[i].drive(); //when i = 5, error will
    be thrown
}
```

Debugging with Printlines

So now you know what type of error it is and where it's being thrown, but you still can't figure out what's wrong. That is totally understandable.

To figure out what is actually going on in your code, you will often have to do the following:

- Figure out which lines of code are actually being executed
- Determine the values of the variables at the time of execution

Both of these cases can be solved using the built-in Java method, `System.out.println(...)`. This method accepts one argument that is of type `String`. When the method is executed, it will print the string to the console (or shell or terminal). However, before we discuss printlines (printlns), you should know a little bit about Strings.

String Boot Camp

A `String` is a built-in Java object that represents text. Strings are very easy to work with.

There are a couple ways to instantiate a string.

You can either create a variable to hold a string...

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

...or you could just create a string on the fly as a parameter:

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

Remember: Strings must be put in quotes.

Also, every single object in Java has a **toString()** method which returns a String that describes the object. Even if you didn't write one in your class, Java automatically creates one for you. Java automatically calls `toString()` on an object when it's being printed out; therefore, the follow lines are synonymous:

```
System.out.println(_car.toString());
```

and

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

You can **concatenate** (a fancypants way of saying combine) strings using the + operator.

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

```
-----<Console>-----
```

```
My name is CS15 TA!
```

That example probably looks pretty useless. You could have created the same String using `String myString = "My name is CS15 TA!"`. The real power, however, comes when you use concatenation like this:

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

```
-----<Console>-----
```

```
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.

Pretty cool right? Strings are very powerful objects. You can take a look at all the other things you can do with Strings by reading the Javadocs:

<http://docs.oracle.com/javase/6/docs/api/java/lang/String.html>

Method Execution

Now that you're a certified graduate from the College of Java Strings, let's see how we can use `println`s to make life nice and easy.

Let's say you are dealing with a mysterious `NullPointerException`. You want to know if a particular method was fully executed before the `NullPointerException` to help you pinpoint the

problem. To determine this you could insert a `println` as the first and last line of the particular method, as demonstrated below:

```
public void drive(){
    System.out.println("execution got to method drive() in
    Car.java");

    //drive() code elided

    System.out.println("reached the end of the method drive()
    in Car.java");
}
```

If the method is entered before the exception but not finished, then when you compile and run your program, you will see the following text printed in your console:

```
execution got to method drive() in Car.java
Exception in thread "main" java.lang.NullPointerException
    at <stack trace elided>
```

But if the whole method is completed, then your console will contain this:

```
execution got to method drive() in Car.java
reached the end of the method drive() in Car.java
Exception in thread "main" java.lang.NullPointerException
    at <stack trace elided>
```

So what? Well, depending on how many `println`s you see--maybe only the first line or both lines or neither--you will know which section of your code is causing the problem.

When you're running into problems and you have no idea which method could be causing the problem, using `println`s should be one of the first things you try. By figuring out which methods and lines are actually being run, you can reduce the number of places you have to look to find the bug. Hooray!

Displaying Values

Another good use of `System.out.println(...)` is to display the values of variables. If you are getting a `NullPointerException`, a good technique is to print out the values of all the variables in that line.

For example, imagine that you've wisely deduced the following line was causing a null pointer exception:

```
_car.move(_grid.getLocation());
```

There are two method calls on two different objects which each could have caused the error in this line. *After looking up what a `NullPointerException` means using the Online TA tool*; you know that at least one of the objects in the line is null. In this case, that would

mean either `_car` is null or `_grid` is null, or both! To figure out what's going on, let's use `println`s. In our code, let's add the following two lines just before the line in question:

```
System.out.println( "Car value: " + _car );
System.out.println( "Grid value: " + _grid );
```

Now run your program. From what displays in the console, you can maybe figure out what was null:

```
Car value: null
Grid value: MyGrid1
Exception in thread "main" java.lang.NullPointerException...
```

Now what? Well, based on this print out, you know that the problem is the `_car` variable. The next step would be to look where it is (or most likely, isn't) assigned a value and fix it. Did you ever instantiate it before you reached this line? Is it a local variable that got garbage collected in some other method?

AN IMPORTANT NOTE ABOUT PRINTLINES:

Make sure to remove all your debugging printlines **BEFORE** handing in your projects!
YOU **WILL** LOSE POINTS if you fail to do so!

Check Point 5: "Debugging" a Program

- Open Eclipse, then open your AndyBot's Maze class.
- Please place the following printlines:
 - Put a printline at the top of your Maze constructor and print out the value of "andy."
 - One that prints that you are "Starting to walk through the maze" just before andy starts to move.
 - A printline in the Maze constructor that states that you are about to call `solveRoadBlock(int x)`
 - A printline that says that you are "Solving the roadblock with password: <password>".
 - <password> should be the value that the method is passing into "enterPassword(int x)"
 - A final printline at the end of the Maze constructor that says that andy has completed his walk.
- **Once you have all the above printlines working, show a TA. At this point, you should also show the TA your written answer from the Design portion of the lab. You may now be checked off for today's lab!**