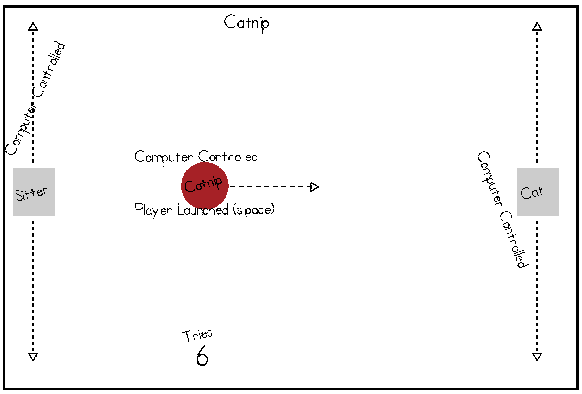
# CIS 201 – Computer Science I
# Laboratory Assignment 2

## Learning objectives

- Creating a class in `emacs`
- Using the keyboard in FANG
- Using variables
- Writing JavaDoc formatted comments
- *What* to comment
- Identifying what code does what
- Looking up features in documentation




## Create a new game called `Catnip`

All Java programs are built using Java *classes*, and all Java classes have names. We normally name a class so that the name generally describes the purpose of the class. Each Java class defines a new Java *type*. We will describe the terms *class* and *type* in more detail as our course progreses. The name of a Java class must begin with an uppercase letter and must consist only of letters, digits, and underscores.

A Java program is simply a text file (that you can create this in `emacs`). The text file has a specific layout that defines the objects and behaviors necessary to carry out some specific defined activity – one that you define or one that someone else has defined and that you are implementing. We call such a file a “source” file, since it is the beginning step (the “source”) necessary to create a running program.

To define a Java class, you need to create a file with the same name as the class but ending with the `.java`. The `.java` part of the name is called a filename *extension*. Other extensions you will likely see in this class and elsewhere are `.class`, `.txt`, `.pdf`, and so forth.

Be sure you are logged in and that you have started a terminal window (shell). Then `cd` into your `CS1` directory. (You should create the `CS1` directory if you haven't done so before.) Then create a new directory with name `Labs` and `cd` into it. Finally, create a new directory with name `Lab02` and `cd` into it. These are the commands you should use:

```
mkdir Labs
cd Labs
mkdir Lab02
cd Lab02
```

Next copy the file `Catnip.java` from the directory `/home/student/Classes/201/Labs/Lab02/Catnip.java` into your current directory with the following command:

```
cp /home/student/Classes/201/Labs/Lab02/Catnip.java .
```

Again, don't forget the dot!

Now start `emacs` with the command

```
emacs Catnip.java
```

When `emacs` opens, it is about 35 lines high. This value was set as part of the initialization of `emacs` so that the windows fit on the monitors in the classroom. You might be more productive if you're able to fill your screen vertically with as much of the `emacs` window that will fit. If you want to change the `emacs` window size, move your mouse pointer to the bottom border of the `emacs` window, click on it with the first mouse button, and pull the window downward or upward to change the vertical size.

You should see the contents of the file `Catnip.java` displayed in your editor window.

Programs should always contain comments. Comments are used to help others who read your code – or you, if you haven't looked at the code for a long time – to understand what the program is about and how it is supposed to work. Comments don't affect the way the program actually works, since comments are essentially ignored by the program compiler.

Java has two ways to include comments: multi-line comments, and end-of-line comments.

Java multi-line comments start with `/*` and end with `*/`. This means that the stuff on lines 3 through 14 of your program are comments. End-of-line comments start with `//` and continue to the end of that line. End-of-line comments are typically much shorter than multi-line comments and normally appear along with program code instead of at the beginning of your program.

The particular multi-line comment starting on line 3 is called a *header* comment and pertains to the whole file. It explains what this class is, what it does, and why it was written. This comment is the first place you or anyone else reading your program would go to find out what the program is about and if this is the right place to fix some problem or to make changes.

So a header comment is written for someone *just like you* – someone who knows just as much Java as you do, so you don't have to explain every little thing about your program, but there should be enough detail so you will know what the program is about. Thus a header is comment is meant to *document your intent*. Since this program implements the game sketched in the diagram at the beginning of the lab, thus header comment briefly describes the game design.

You should replace the `date` part with today's date and the two lines referring you and your lab partner with your correct email addresses and names.

## Add variables

The Cantip program needs three sprites as game components, two `RectangleSprite`s and one `OvalSprite`.

As we discussed in class, a variable is declared by specifying its visitibily, its type and its name – in that order. A variable must be declared inside of a block (between two curly braces); the variable is generally *visible* anywhere within the block where it is defined – including in sub-blocks defined within that block. Don't worry about this right now – we will re-visit these terms in class.

Our three sprites will be used throughout the Catnip game, so they will be declared inside the `Catnip` class.

Naming variables is important: you want to use meaningful variable names. Java is case sensitive in naming and permits a letter followed by any number of letters, underscores, or digits for a valid name. Variables should start with a lowercase letter.

One of our variables represents a cat, one a cat sitter, and on a bag of catnip. So this is what you need to type just after the `Catnip` class declaration, after the first curley brace. Put these above the other variable declarations already there.

```
private RectangleSprite catSitter;
private RectangleSprite cat;
private OvalSprite catnip;
```

You should try to compile the code, using the `JDK -> Compile` menu. You will find that the program will not compile. The fire complaint will be that the compiler can't find the symbol `RectangleSprite`. This is because your program doesn't know about `RectangleSprites`. The same thing applies later in your compiler error statements with regard to `OvalSprites`.

You will need to add `import` statements for the types of these variables, So, *at the top of your program*, after the `fang.core.Game` line, add the following lines:

```
import fang.sprites.RectangleSprite;
import fang.sprites.OvalSprite;
```

Now compile your code and notice that these errors have been fixed.

### Checkpoint 1

**Show us that your code compiles correctly at this point.**

## 1 Examine the `setup` method

The `setup` method is part of the FANG library. Go to your moodle account and bring up the FANG documentation. (You can also get to this documentation from our department homepage, `cs.potsdam.edu`, under `Documentation`.)

The documentation comes up in a window that has a big list of lots of things. We are interested in the class `fang.core.Game` so click on one of the `fang.core` links to limit the view to that package. Click on it. This provides the documentation generated from comments in that class – you might notice that the FANG documentation is pretty sparse, but this doesn't mean that *your* documentation should be as bad.

If you scroll down the page, you will find that `setup` returns the `void` type and has an empty parameter list. We have to remember that the method must be `public` for the game to call it. So our method signature is for `setup` looks like this:

```
public void setup()
```

Looking at the `fang.sprites` page of the documentation, the `RectangleSprite` constructor takes a width and a height for the sprite. Notice that our `RectangleSprite` object has been created – using the `new` operator – with width 0.10 and height 0.10. **The game's drawing canvas is always assumed to be width 1.0 and height 1.0. The top-left corner of the canvas has coordinates (0,0), so that the horizontal ($x$) coordinates go from left to right, and the vertical ($y$) coordinates go from top to bottom.**

The `setup` code also sets the speeds of the sprites. **These speeds are measured in units of *screens per second*.**

The `advance` method is what makes the compontents move: both the cat and the cat sitter move vertically (only their *y*-coordinates change) and bounce off the bottom and top of the screen. The catnip moves from left to right across the screen. If it is "caught" by the cat, it's a *hit*. If not, it's a *miss*.

### Checkpoint 2

**Show us your running game at this point.**

## Changing the speed parameters

You can change the speeds of the cat, the cat sitter, and the catnip. Experiment with these values and see what happerns. What if you set the cat sitter speed to zero? What if you set the catnip speed to zero?

### Checkpoint 3

**Show us some of your changes.**

## Keeping track of the score

Declare a `score` variable along with your other variables like `cat`. What should be the visibility of this variable? What should be its type?

In your `setup` method, initialze the `score` to zero, similar to setting the `catnipSpeed` to zero. The difference is that the `score`'s initial value should be an integer, not a decimal number. Add the appropriate line to do so. Also, un-comment the line in the `advance` method that updates the score. Finally, un-comment the line in the `updateScore` method that displays the score using `System.out.println`.

### Checkpoint 4

**Show us your changes, and show us where the score is being displayed!**

## Keep track of launched catnip

Create a new variable `launched` that will keep track of the number of catnips that have been launched. Find the proper places to declare your variable, initialize it, and update it. When each catnip is launched, also call `updateScore`. Change `updateScore` to print both the number of catnips launched and the score.

### Checkpoint 5

**Show us your changes.**