Computer Science 225
Advanced Programming
Siena College
Spring 2022

Topic Notes: More Threads and Animation

Repaint Frequency

A few of you have noticed that we are sometimes creating a large number of threads in our programs. Fortunately, Java threads are a pretty lightweight constructs and we can reasonably create a large number. This is especially true for the kinds of threads we have used so far, where threads spend most of their time in a sleep call.

One thing those threads often do in our graphics programs with animation is to call `paintComponent`. If we have dozens or hundreds of threads, each calling `repaint` periodically, it could end up resulting in many more calls to `paintComponent` than are necessary for a smooth animation. Since we can only see 24 frames per second, there is no need to refresh the window at a rate much higher than that.

We can see how many times `paintComponent` is being called by introducing a `static` variable and displaying its value.

During class, we will modify the `SnowScene` program in two ways:

1. Threads responsible for individual animated objects (in this case, the `FallingSnow` objects) will update the positions of those objects, but not make a call to `repaint`.
2. An additional `Thread` object will be created that simply calls `repaint` about 30 times per second. Since we never need to interact with this object other than to call its `start` method, it can be constructed as an anonymous class.

Think about the circumstances where this version will result in fewer `paintComponent` calls and circumstances where it will result in more. Can we make any changes to limit the latter?

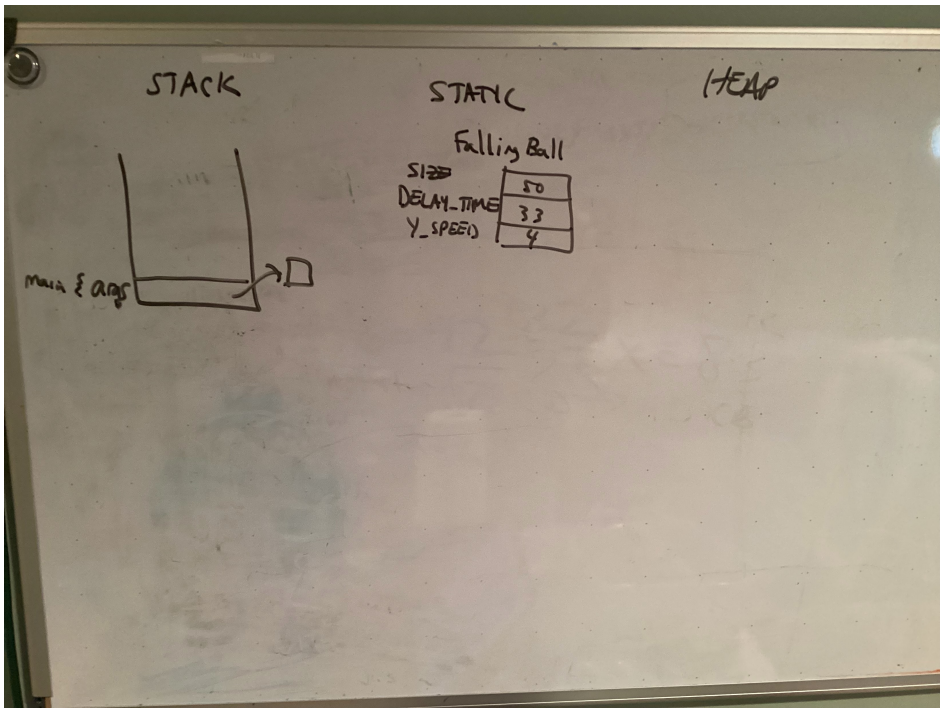
Memory and Threads

Multithreading and event-driven programming each add a significant level of complexity over single-threaded execution. It is important to have a clear understanding of what objects exist, what methods are in execution (and hence have their parameters and local variables on a stack), and which thread will be executing each method.

To help understand all of this, let's look back at the `FallingBallDemo` example.

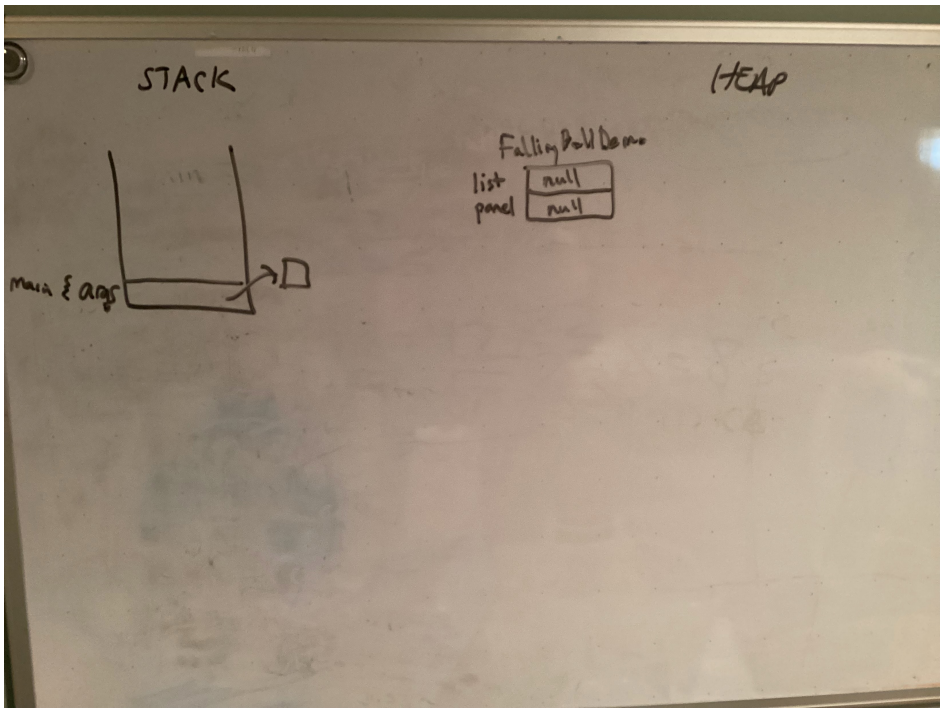
As with all Java applications, we begin with two steps. First, the `main` method's activation record is placed on the stack. An activation record consists of the formal parameters and local variables of

the method. In this case, there's just the `args` parameter, which we are not actually using. Second, any class (`static`) variables are placed into static memory. We draw these with each class that includes any class variables. Remember, class variables exist exactly once even if no instance of the class is ever created, and no matter how many instances are ever created.



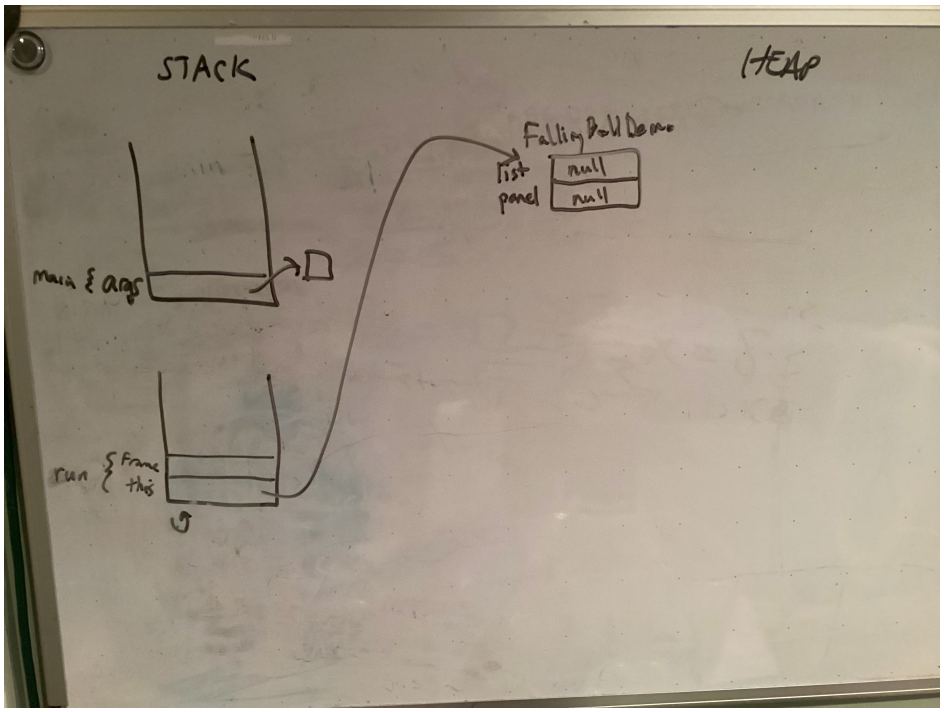
To simplify the diagram going forward, the static variables will not be shown here, but they do continue to exist.

Now, the `main` method executes. The first thing it does is to construct an instance of the `FallingBallDemo` class. As with any object, this results in a chunk of memory being allocated on the heap, large enough to hold all of the instance variables of the class.

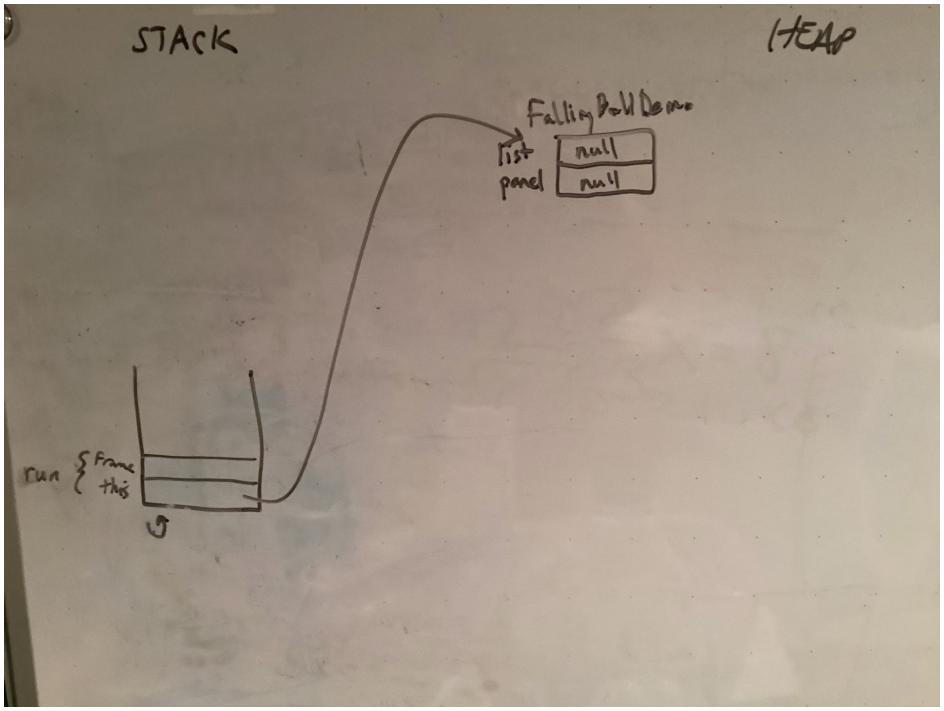


If that class had a constructor (this one doesn't), the constructor's activation record would be placed on the stack, and the constructor would execute. Upon completion of execution, the activation record would be removed from the stack.

The object construction returns a reference to the object just created. In this case, it just gets passed to the `invokeLater` method. This method will create a new thread of execution, and in that thread, call the `run` method of the object (which must be an instance of a class that implements the `Runnable` interface, so we know it would have such a `run` method). It is critical to note here that the `main` method is running in the original thread, while the `FallingBallDemo` `run` method will begin to execute in a new thread, which has its own call stack. The `run` method's activation record is placed on this stack. It will contain all parameters (it has none), local variables (it just has one), and, since it's an instance method, will have a `this` reference to the object whose instance variables it can use.

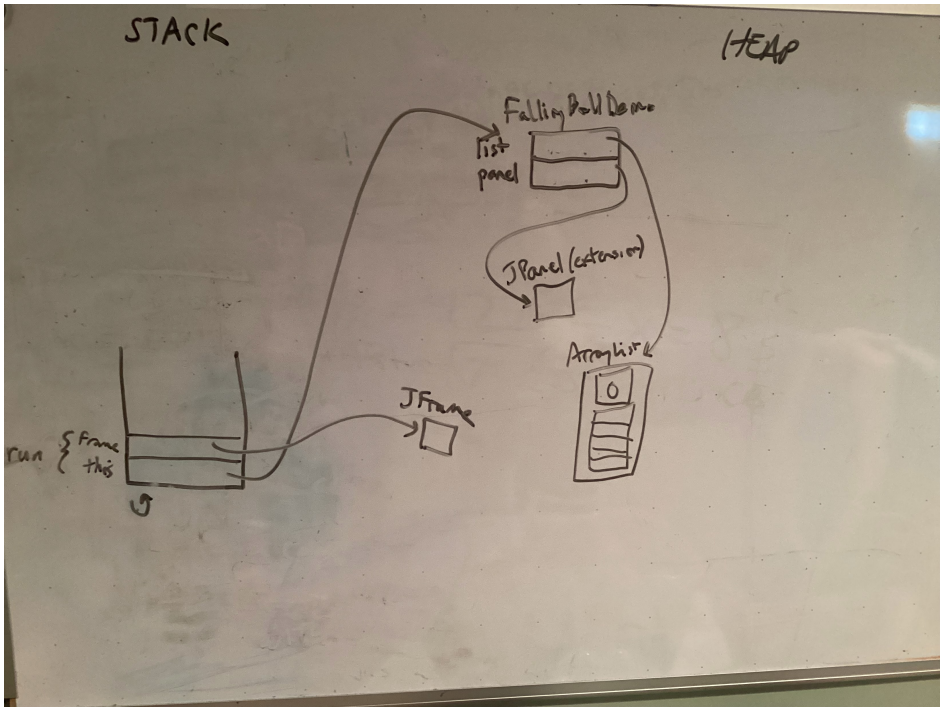


Now, two things are happening. `main` is completing its execution, ending the life of the original thread. So not only does `main`'s activation record get removed from the original call stack, but that stack will also go out of existence.

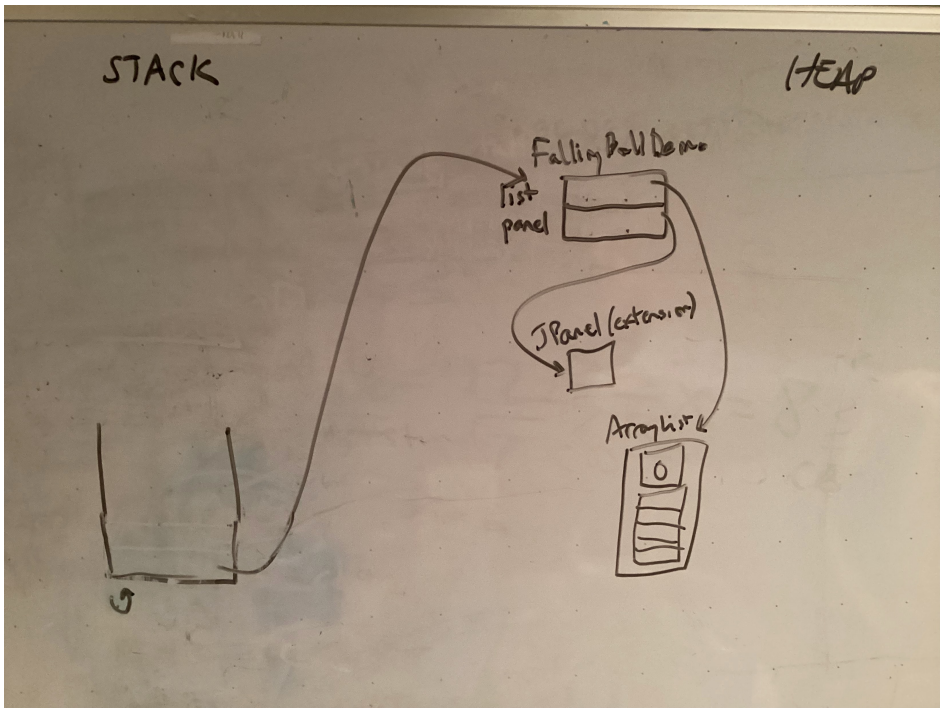


Meanwhile, the `FallingBallDemo` `run` method is executing. It does several things, a few of which are represented on the diagram. A `JFrame` is constructed and a reference to it is stored in

the `frame` local variable. The anonymous extension of `JPanel` is constructed, and a reference to it is stored in `panel`, and an `ArrayList` is constructed, and a reference to it is stored in `list`.

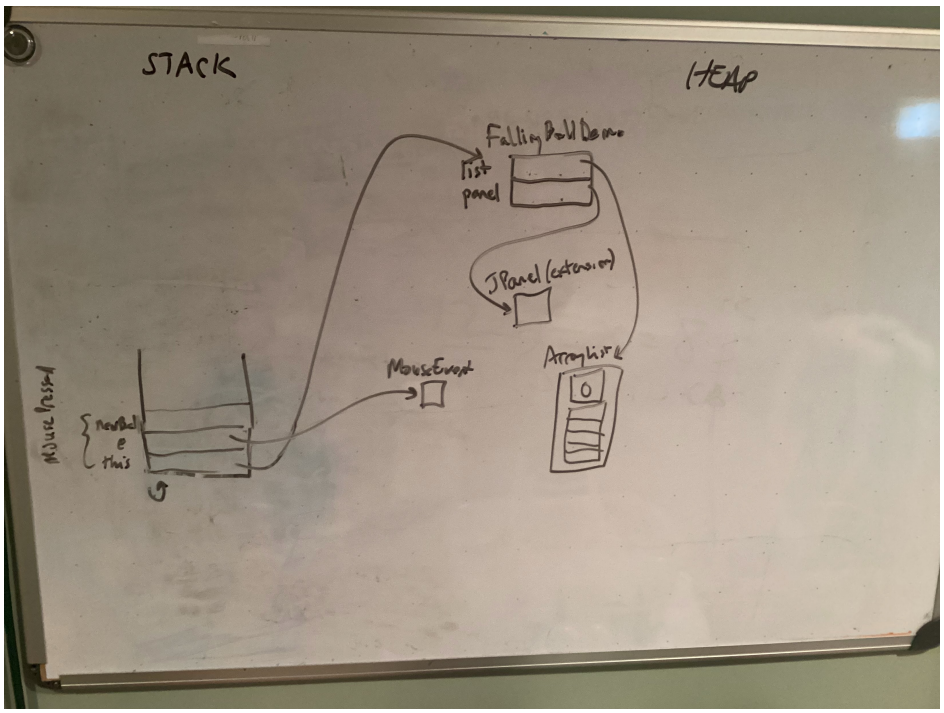


When the `run` method finishes, its activation record is removed from the stack, and we also lose our only reference to the `JFrame`.



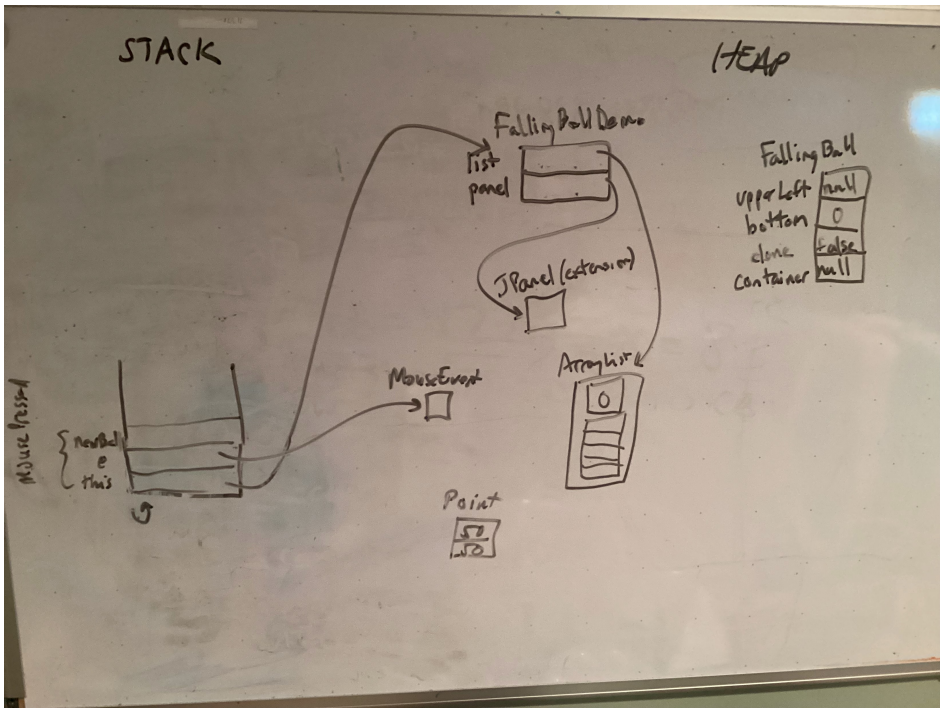
Now, nothing happens until a mouse event triggers a call to `mousePressed`, or a repaint event triggers a call to our anonymous extension of `JPanel`'s `paintComponent` method. Either of those would result in that method's activation record being placed on the call stack. For the remainder of the program, that thread is dedicated to executing event handlers. Fortunately, it is not our concern how they get called, but we should understand what happens when they do.

So, let's assume the mouse is pressed at coordinates (50,50). This will cause a mouse press event, and a call to `FallingBallDemo`'s `mousePressed` method in the event handler thread. The activation record for `mousePressed` will be placed on the stack. Since it is a method of our one instance of `FallingBallDemo`, its `this` reference will refer to that object.

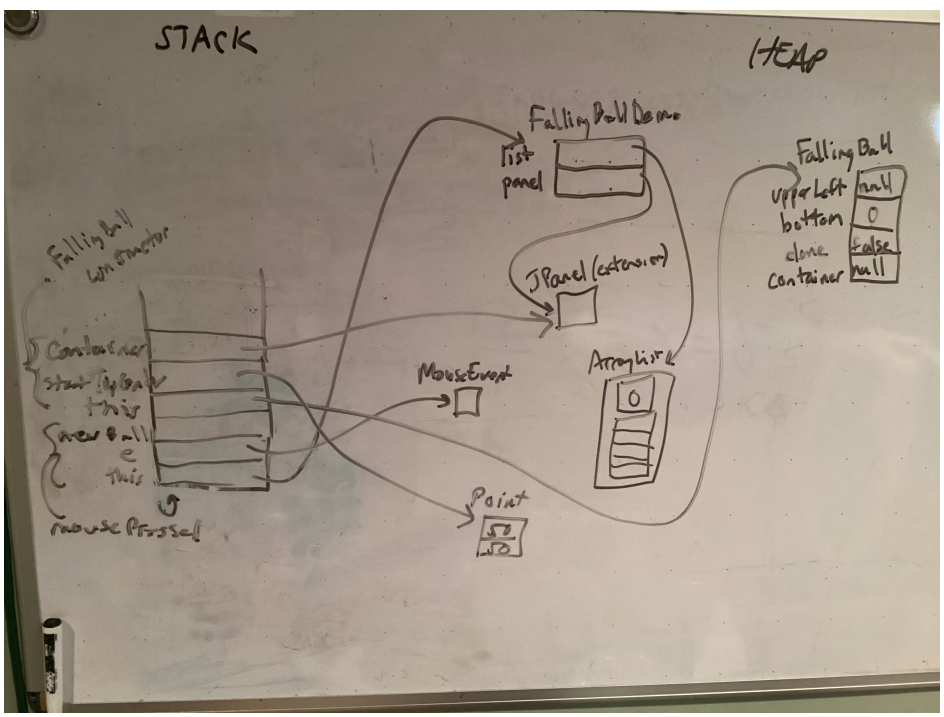


We have also drawn the `MouseEvent` object that is passed to our method, and which will be used to get the point. Now `mousePressed` execution begins. The first line has many things going on. The call to `e.getPoint()` will give us a reference to a `Point` object containing the coordinates (50,50), and that will be passed to the `FallingBall` constructor.

The `FallingBall` construction follows the usual process for any object construction – it only becomes different because it's an active object when we will later call its `start` method. First, a chunk of memory large enough to hold a copy of all of `FallingBall`'s instance variables is created on the heap. They are initialized to `null`, 0, and `false` values, as appropriate.

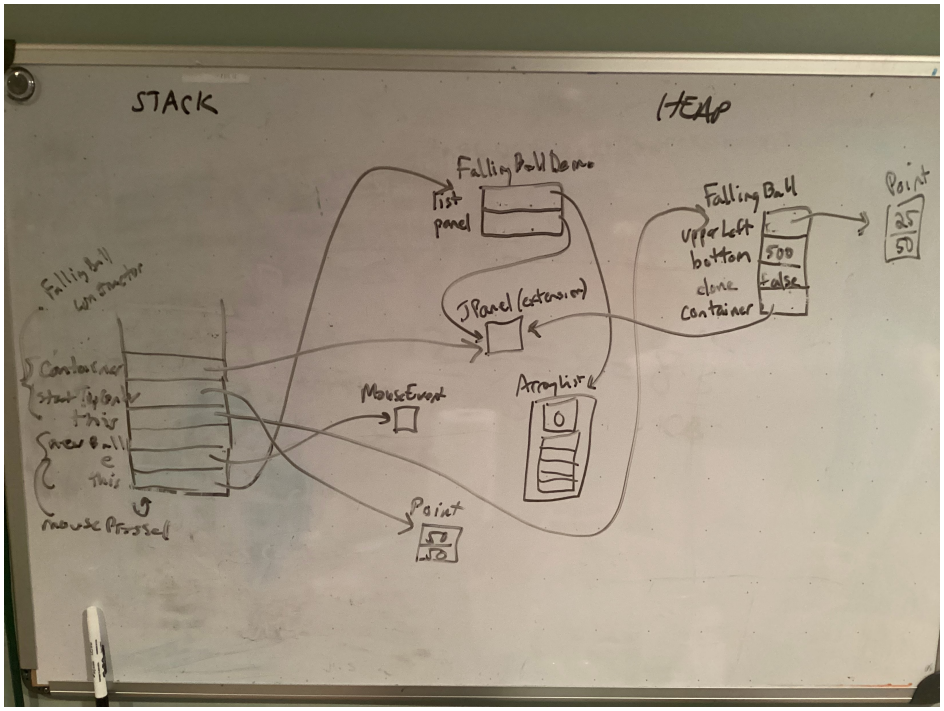


The `FallingBall` constructor is called, executed by our event handler thread. The activation record is placed on that thread's stack. Its `this` reference points to the `FallingBall` object being constructed, and the two formal parameters are initialized to the values passed as actual parameters in the constructor call.

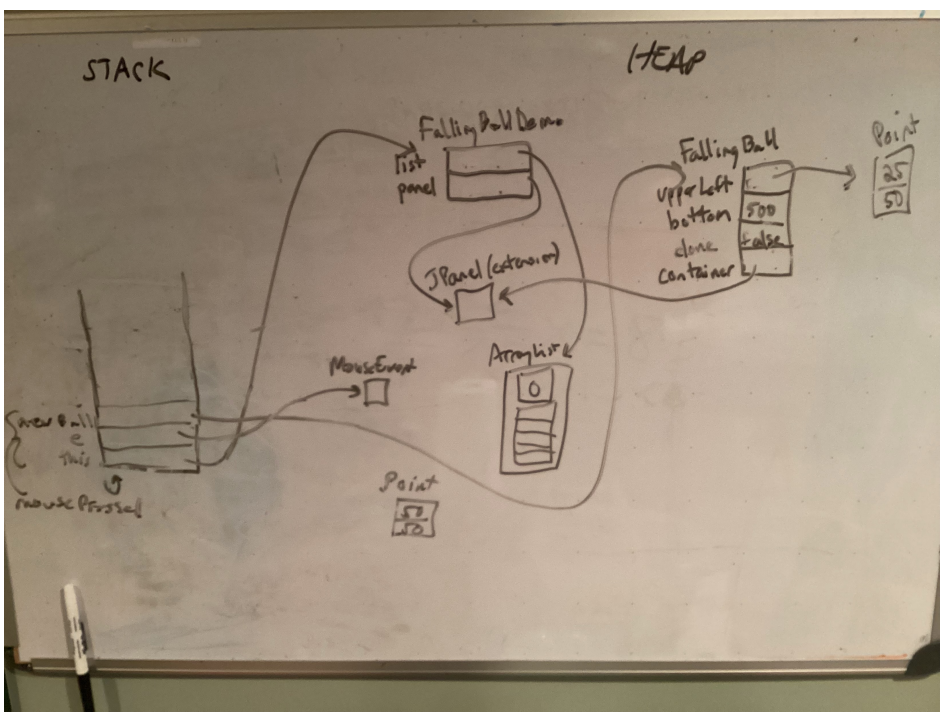


The `FallingBall` constructor executes, constructing and saving a reference to a new `Point`,

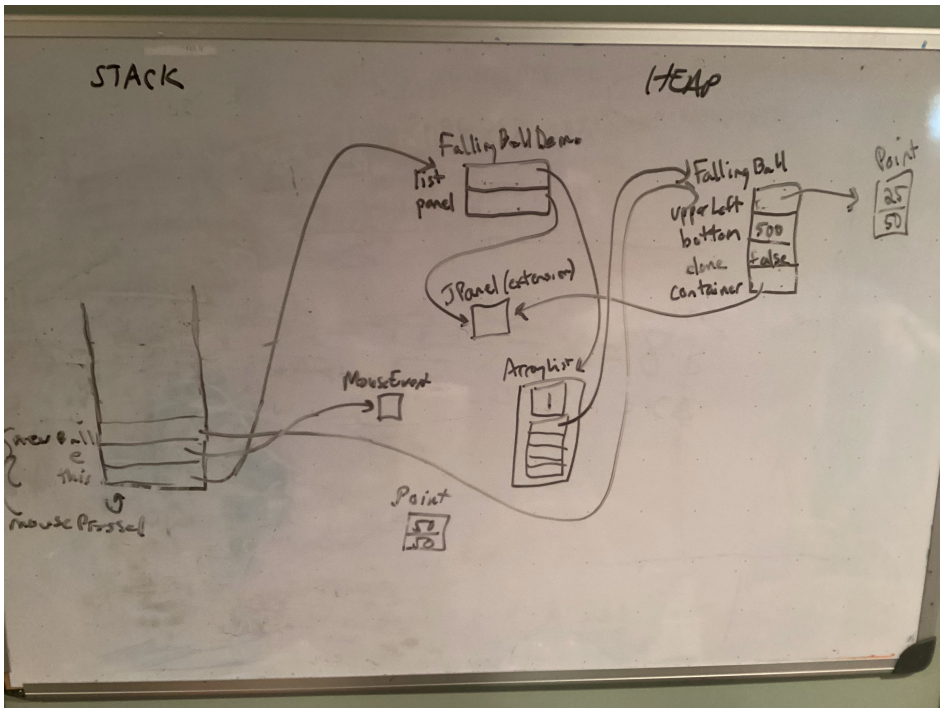
and storing two other values in instance variables.



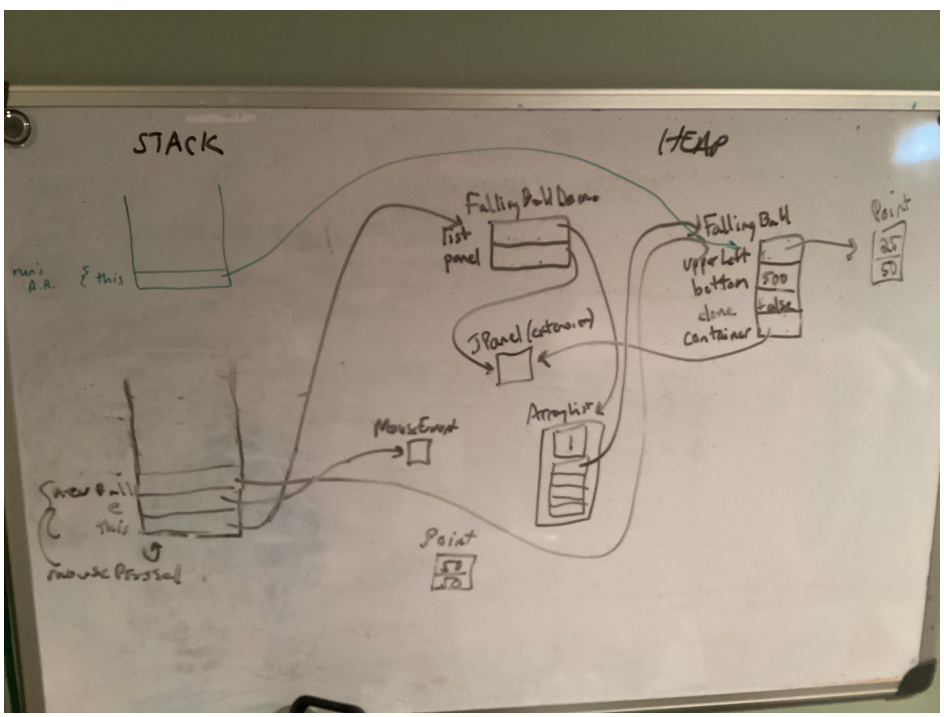
The constructor finishes, and its activation record is removed from the stack, and the reference to the new object is stored in the local variable `newBall` in `mousePressed`.



The reference `newBall` is added to the list.



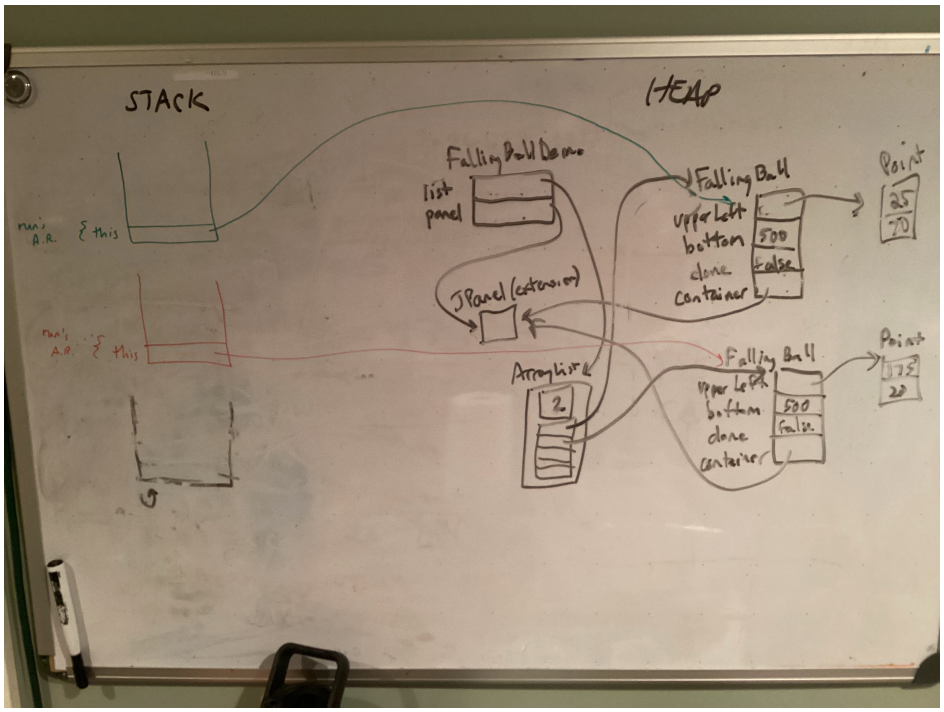
The call to `newBall`'s `start` method is what creates a new thread for the execution of its `run` method. Once this happens, we have a new stack where the thread object's (in this case the `FallingBall` we just constructed) `run` method will execute, concurrently with our existing event handling thread.



Meanwhile, the `mousePressed` method continues to execute in the event handling thread, call-

ing `repaint`, which will cause a repaint event. At some point in the very near future, that will cause the `paintComponent` method to execute. But for now, the `mousePressed` method completes, and its activation record is removed from the stack.

If we press the mouse again, say, at (200, 20), the whole process of the `mousePressed` method, including the construction of another `FallingBall` object, and a new thread to execute its `run` method, will complete. Assuming the first `FallingBall`'s `run` method has had time to move the ball down the screen a bit, the new situation would look like this (after `mousePressed` has completed, and if no `paintComponent` method call is currently executing).



At this point, there are three threads in the program. The event handling thread will execute any mouse or paint event handlers. And each other thread will execute the `run` method of one of the `FallingBall` objects.

Each subsequent mouse press would result in another `FallingBall` being added to the `ArrayList` and have a new thread created to execute its `run` method. As each `FallingBall`'s `run` method completes and returns, its thread would cease to exist, and calls to that `FallingBall`'s `done` method would return `true`. This will result in that `FallingBall`'s reference being removed from the `ArrayList` during the loop in a subsequent call to `paintComponent`.

A Bouncing Ball Animation

In our next example

`https://github.com/SienaCSISAdvancedProgramming/BallTosser`

we see yet another animated ball object, with this one having a few new features:

- By dragging the mouse, a new ball can be launched on different trajectories and speeds.
- The ball bounces off the walls and floor, losing a bit of its energy each time it does so.
- The ball is subject to gravity.
- The ball disappears when its motion gets close to 0.

The launching of the ball is done similarly to how you launched the ball in Ski Ball. The differences in the x and y coordinates of the press point and the release point, subject to a scaling factor, are used to determine its initial speed in each direction.

The motion in y is subject to a gravitational pull, like in some earlier work.

When the ball hits any side of the window, it bounces off. We simulate it losing some energy on bounces by multiplying the speeds by a dampening factor any time a bounce occurs.

We decide the ball is done moving when it is on the "floor" and its motion in both dimensions is close to 0.

Another Look at Thread Safety

We have been working with animations for a while, and have only briefly talked about the dangers of concurrency. Those going on to take Operating Systems will study these issues in much more detail. Our concern here is to recognize when *thread safety* is a potential issue, and to learn how to ensure that we don't run into problems.

We want to look for situations where the same data might be used by two threads at the same time, and at least one of those threads might be modifying that data. We saw that can happen even with very simple data when we looked at the two threads that concurrently modify a `counter` variable with increment and decrement operations.

Along similar lines, consider an example of a singly-linked list data structure, where a reference to the first node in the list is stored in an instance variable `head`. Our program creates a list `x`:

```
SimpleLinkedList<Integer> x = new SimpleLinkedList<Integer>();
```

The program then creates two threads, each of which tries to add a value to this (so far) empty list:

Thread A:

```
x.add(1);
```

Thread B:

```
x.add(2);
```

We would expect that after these two statements execute, our list would contain two values, 1 and 2, in either order. Whichever thread executed its `add` method later would place its value at the start of the list.

Recall that the code for the `add` method of such a list has a case for adding at position 0, which is what the method calls above are trying to do:

```
 if (pos == 0) {
    head = new SimpleListNode<E>(obj, head);
    return;
}
```

Suppose both `add` calls happen at almost exactly the same time. Both calls go into the `if` statement with the value of `head` being `null`. Both call the `SimpleListNode` constructor, with that `null` reference as the second parameter. Both constructor calls return a new node with a `null` value for their `next` reference, and both assign a reference to the just-constructed node to the `head` instance variable of the list.

What's in the list? We'd end up with just one node, and it would contain the value added by whichever thread assigned the `head` variable last. This is another example of a race condition.

We can avoid this kind of problem by declaring the methods of the list with the `synchronized` keyword, so at most one thread can be executing any method of the list class at any given time. This would work in this case to ensure thread safety. However, there are potential problems remaining, both related to the *granularity* of synchronization we are using.

- Placing the `synchronized` keyword on the method headers will serialize **all** access to the methods with that keyword, not just in situations where the instance variables of the list and its nodes are being modified. This reduces our concurrency and limits the ability of our threaded implementation to execute efficiently, especially when trying to take advantage of multiple processing cores. Further, the underlying implementation that ensures proper behavior of `synchronized` methods comes with computational costs. Every Java object has what is called an *intrinsic lock* that is used to support exclusive access for synchronization.
- This does not handle cases where the problem arises from unfortunate thread interactions that span multiple method calls. That is, the problem arises from the interleavings of the method calls themselves, not the code within a method.

Note: as has been mentioned a few times this semester, the first item above is the difference between `java.util.Vector` and `java.util.ArrayList`. All methods of `Vector` are `synchronized`, while `ArrayList`'s are not. So when doing single-threaded programming, we tend to use the `ArrayList` to avoid the overhead.

Java has a mechanism to take any class that implements the `List` interface and wrap it up so that all of the methods are `synchronized`. The `static` method `Collections.synchronizedList` is provided for this purpose. There are similar methods for other Java API collection interfaces including `Collection`, `Map`, and `Set`.

Ultimately, we want to make the `synchronized` code segments as small as possible so as not to limit concurrency, but large enough to ensure thread safety in the given situation.

The latter case from above can be demonstrated again with a linked list (though this would work just as well for something like an `ArrayList`).

We have a list with one element:

```
SimpleLinkedList<Integer> x = new SimpleLinkedList<Integer>();
x.add(1);
```

Then two threads each try to remove a value to make use of it in some way:

```
if (!x.isEmpty()) {
  int val = x.get(0);
  x.remove(0);
  // do something
}
```

What is the intended behavior here? What could go wrong? (in-class exercise)

This is the kind of problem that can occur in many of our animation examples, so we'll use one of those, `BallTosser`, to demonstrate a way to handle it.

What we need to do is to ensure *mutual exclusion* on the parts of the code that cannot safely execute concurrently. These parts of the code are called *critical sections*.

We introduce a new instance variable of type `Object`, which will be used as an *explicit lock*. We place the code in any method that forms our critical sections, for which we want to ensure mutual exclusion inside a block that looks like this, for an explicit lock named "`lock`":

```
synchronized (lock) {
   // critical section code
}
```

In the `BallTosser`, we create an explicit lock as an instance variable, then need to place two segments of code in the `synchronized` blocks. These are the two parts of the code that can modify our list: the loop in `paintComponent`, and the call to `add` in `mouseReleased`.

The modified version can be found in

`https://github.com/SienaCSISAdvancedProgramming/SynchronizedBallTosser`