SIENA*college*
Computer Science

Computer Science 225
Advanced Programming
Siena College
Spring 2020

# Topic Notes: Threads and Animation

As experienced programmers, you know that loop constructs can be used to get a set of statements to be executed repeatedly. The iterations of the loop happen one after the other, and are done as quickly as the computer can execute them.

In our work on event-driven programming, we have seen that statements can be executed repeatedly in response to events. There are also programs that are executing multiple segments of code simultaneously. It is Java's support for this kind of programming, through *multithreading*, that we will be studying next.

---

## Threads to Support Animation

All of the classes we have defined so far have described "passive" objects. They only do things when they are told to (*i.e.*, because someone invokes one of their methods).

Our first such example is

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

in which a new circle (a ball) is created on the screen each time the mouse is pressed, and that ball "falls" slowly down until it goes off the bottom of the window.

Before we look at the details of the implementation, let's think about how a program that can accomplish this might work. We know that we will need to handle mouse press events to create the ball. But how do we get the ball to move at an appropriate speed and have the window be repainted each time it moves?

One might think about having a loop in the `mousePressed` method that repeatedly moves the ball a little and repaints. Unfortunately, this would fail for a few reasons. First, the loop would execute so quickly that we would never have chance to see the ball. A delay loop (a loop that does nothing useful, intended to slow down execution) could be used. However, the paint event that would result in the `paintComponent` method being called to update the screen would not be called until the mouse event handler returns. So we would never see anything. Even if we were able to get the screen to update during the mouse event handler's execution, subsequent mouse events would also not be processed until the first had completed.

The essential problem here is that our program's `run` method, then the subsequent event handlers (including `mousePressed` and `paintComponent`) execute in a single *thread of execution*, often referred to simply as a *thread*. We need to introduce additional threads so our program can have multiple parts of the program executing *concurrently*. This is what our `FallingBallDemo` example does.

Here are some key points to notice about this example:

- The `FallingBallDemo` class is mostly familiar from our earlier examples.
- `FallingBallDemo` maintains a list of `FallingBall` objects much the way our `DragMany` example has a list of shapes to draw. The `mousePressed` method creates new ones and adds them to the list, and the `paintComponent` method traverses that list and calls each `FallingBall`'s `paint` method, so it can draw itself at the appropriate position. If the ball has finished its descent, it is removed from the list so it is never drawn again.
- Like the `DraggableShape` objects in `DragMany` example, the `FallingBall` objects maintain the information necessary to be drawn.
- The `FallingBall` class extends the `Thread` class, meaning that it will be able to execute on its own: we can think of it as *having its own "brain"*.
- That "brain" will spend its life executing its `run` method. In `FallingBall`, the `run` method consists of a `while` loop.
  - The loop continues as long as the top of the ball is above the y-coordinate of the bottom of the window.
  - Each iteration of the loop begins with a 33 ms `sleep`, an amount chosen so that the loop will execute about 30 times per second.
  - On each iteration, the coordinates of the ball are moved down by a small amount, and the `repaint` method is called so the new position will be drawn by the panel's `paintComponent` method.
- When the loop terminates, the `done` variable is set to `true`, so the `done` method will return `true`, triggering the object to be removed from the list of `FallingBall`s instead of being drawn in the `paintComponent` method.
- The `run` method does not get called directly, but instead it is invoked within the new thread by a call to the object's `start` method. This is done in `mousePressed` immediately after the object is created and is added to the list.

---

## Threads Creating Objects

In the previous example, we saw that by making an object a thread, it can control its own motion. However, it was the `mousePressed` event handler that caused new objects to be created. Our next example,

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

continues the use of threads to control animation of objects, but also introduces threads that are responsible for creation of new objects over time.

Before we get into the threads aspect of the `SnowScene` program, let's take a quick look at Java's mechanism to draw images with our graphics primitives. The same example includes a small program `ShowSnowflake` that simply loads a graphics file (in this case, a small transparent GIF image of a snowflake) into a static `Image` object in its `main` method, which can later be passed to the method in `paintComponent`.

Normally, that's enough, but in this example, we wish to draw the image immediately on the window's creation, and it's possible the `Image` object has not yet been fully created. This is because the `getImage` method call in `main` is *asynchronous*. This means that it returns immediately, even though the `Image` object returned does not yet contain all of the information from the file. If we call `drawImage` before the asynchronous operation to read the contents of the file and put the data into the `Image` object has completed, the `drawImage` call will fail and return `false`.

Java's mechanism to account for this is to pass the last parameter of the `drawImage` method an instance of a class that implements the `ImageObserver` interface. In this example, the `ImageObserver` is the class itself (as we've been doing with our mouse and other event handlers). If the `imageUpdate` method is called, it means that the `drawImage` call failed and we are getting an update on the status of the asynchonous file read operation. If the `infoflags` parameter has a certain bit set (defined by `ImageObserver`.ALLBITS), it means the reading is complete. In our example, we use this information to trigger a `repaint`, so the `paintComponent` method will be called again and the image drawn successfully, and return `false` to indicate that we have received the information we need. If `imageUpdate` was called for other reasons (based on the `infoflags` bits), we return `true` to indicate that we would like further updates.

Back to the main part of this example and the `SnowScene` class.

- We again load in the image in the `main` method, but it is done by calling a `static` method of the `FallingSnow` class, since that is the class that will need to use the `Image` in calls to `drawImage`. In this case, the image is not needed immediately and we are calling repaint many times, so the `ImageObserver` code is omitted.
- A simple background image is drawn in `redrawScene`, called from `paintComponent`.
- Moving to the `mousePressed` method, a new `Cloud` object is constructed and added to a list on each press. Since `Cloud` extends `Thread` (and we will look very soon at the details), we call the new `Cloud`'s `start` method to activate the thread so the `run` method we provide starts to execute.
- Back in `paintComponent`, there is a loop to call the `paint` method for each `Cloud`.
- `Cloud` itself does not draw any graphical objects directly. Instead, it maintains a list of `FallingSnow` objects, which are constructed and added to the list every 900 ms.
- `FallingSnow` is very similar to the `FallingBall` we looked at earlier. The differences:
  - Each `FallingSnow` starts with an x-coordinate chosen randomly by the `Cloud` that creates it, and a y-coordinate so it is just off the top of the window.
  - It falls until it reaches the bottom of the window, then stays there for 2 seconds before "melting".

  - Its `paint` method uses `drawImage`, but only if it has not melted yet.

- The `FallingSnow` objects are never removed from their `Cloud`'s list, so their `paint` methods continue to be called even after they have melted.

- `Cloud` objects, however, are removed from `SnowScene`'s list of `Clouds` when they have finished generating `FallingSnow` objects and all of those `FallingSnow` objects have melted.

---

## Dangers of Concurrency

---

## Interference

We've talked about how an object that extends `Thread` activates an extra "brain" for your program. Imagine your body was controlled by two brains. Suppose one brain gives instructions to your muscles to start running, the other to sit down. The muscles will receive multiple messages, and the result is not likely to be good. Back to "brains" in our programs, what happens if those brains give your program contradictory messages?

An important class of problems can arise with concurrency when there are several threads that might try to update the same variable at the same time. Many of the important problems with parallel, concurrent, and distributed programs boil down to this problem.

Consider an example of a bank with two ATMs which can be used to deposit and withdraw money.

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

We start by looking at the version in the "Danger1" subdirectory.

One of the ATMs will repeatedly withdraw $100 from the account while the other will repeatedly deposit $100 into the account (see the difference in parameters in the constructors for the ATM's). When the user pushes the button, the `actionPerformed` method repeats the construction and execution of the ATM objects.

The main items of interest here are the `getBalance` and `setBalance` methods. They do the obvious things.

The `run` method repeatedly deducts `change` from the account by first executing

```
account.getBalance()
```

to get the balance and then executing

```
account.setBalance(balance+change)
```

to update the balance.

The final balance in the account should be $1000, the same amount started with, as one of the ATM objects withdraws $100 ten times, while the other deposits $100 ten times. If you run this code enough, however, you will discover that the answer does not always turn out to be $1000! What is causing the problems? Look at the program to see if you can determine what is going wrong before reading further.

The error occurs because two different threads (objects of type `Thread`) are updating the same variable, `balance`. Each gets the balance from the bank, adds in its change, and then tells the bank the new balance. However, it can happen that both ATMs get the balance before either of them has the opportunity to update the balance.

For example, suppose ATM1 gets the balance of $1000, while ATM2 "simultaneously" gets the balance of $1000. They might not occur truly simultanously, especially if there is only one processor, but for our purposes it can be helpful to think that way, and most modern computers have multiple cores that can execute code truly concurrently. Now ATM1 adds $100 to the balance and updates the balance to $1100. ATM2 then subtracts $100 from the balance that it originally got ($1000) and updates the balance to $900. Thus if the *interleaving of operations* of ATM1 and ATM2 are such that both get balances before either registers the new balance, the final balance will not reflect one of the two operations. This is called *interference* and is an example of a *race condition*, as the two threads are essentially racing each other to query and update the balance, and whoever updates last has their value remain.

Clearly this is a problem, yet we would like to have the operations of the two ATMs interleaved. (We could just run ATM1 to conclusion before starting up ATM2, but this does not model the usage of ATM's properly.)

We would like to ensure that if ATM1 queries the balance with the intent to change it and set a new balance, that ATM2 does not read the original balance. It is when both read the old balance and both update that one of the transactions is lost. We attempt to remedy this by replacing the `setBalance` method with a `changeBalance` method.

This is in the "Danger2" subdirectory.

Now rather than having separate methods to get and set the balance, we have a single method which takes the amount of change and updates the balance. Because the getting and setting are no longer separated by distinct method calls, the chances of interference are not as great. However there is still the opportunity of interference between the calculation of the new balance and the update of the value. We have artificially increased the chances of this by adding the `sleep` between the two.

Even if we remove that, we reduce the time between the calculation of the old balance and setting of the new balance, but still allows the (at least theoretical) possibility of interference between the calculation of `balance+change` and the assignment of that value to `balance`. To be absolutely safe, we must ensure that only one thread at a time can execute the method `changeBalance`. In general, this idea is called *process or thread synchonization*. We can do this in Java by using the keyword `synchronized`.

If we attach the keyword `synchronized` to one or more methods in a class, then Java will ensure that only one thread at a time will be executing any of those methods. For example we can label both `getBalance` and `changeBalance` as `synchronized`.

This is what you find in the "Safe" subdirectory.

Now if a thread associated with one ATM object is executing either of these methods, then no other thread can execute either of the methods. For example, if `ATM1` is executing `changeBalance`, then `ATM2` will not be allowed to execute either `changeBalance` or `getBalance`. Instead it will wait until `ATM1` has finished executing that method and then execute the desired method. (The operating system is given the responsibility of scheduling the threads' access to the processor.)

A thread executing either `changeBalance` or `getBalance` has no impact on another thread's attempts at executing any of the non-synchronized methods of the program. Thus the user-interface thread can by executing the `actionPerformed` or `startATMs` method while `ATM1` is executing `changeBalance`.

Because of the use of `synchronized`, neither thread can interfere with the other, ensuring that the final answer is the correct one. However, there is one disadvantage of using synchronized methods – they cut down on the amount of concurrency in the system. This may slow down the execution of the program, as one thread may be waiting for an operation to complete (e.g., a write to the screen or a read from a file), while another might be ready to do something. The second thread may be ready to use the processor, but if it is ready to execute a synchronized method and the other thread is executing a synchronized method of the same object, then it may be blocked from executing.

This example may seem a bit contrived – we carefully made sure the sleep times for the two ATMs were the same and added a random sleep inside the methods that change the balance to increase the chances of interference. However, the interference could happen in each of our cases (except the one with the `synchronized` modifiers) even without the careful attempts to increase the chances. Has anyone ever had some big program, even maybe a commercial program, crash in an unexpected and non-reproducable manner? Perhaps a mobile app, a browser, or even an operating system itself? There's a pretty good chance that a lot of those kinds of crashes are the result of concurrency not being dealt with carefully enough. Most of the time, things are fine – but once in a while just the right combination of things is happening and there you go. Crash and burn, and in the worst case, read that literally.

Bottom line, when you have multiple threads that could be executing code in the same class concurrently, methods that use an instance variable could potentially be problematic, even for seemingly very simple operations.

Suppose such a class has an instance variable `counter`. One thread executes

```
counter++;
```

in some method and the other thread executes

```
counter--;
```

in the same or another method concurrently. To see the danger, let's think about how a computer actually executes those statements to increment or decrement `counter`.

`counter++` really requires three machine instructions: $(i)$ load a register with the value of `counter`'s memory location, $(ii)$ increment the register, and $(iii)$ store the register value back in `counter`'s memory location. There's no reason that the operating system can't switch the process out in the middle of this.

Consider the two statements that modify `counter`:

| Thread A | | Thread B | |
|---|---|---|---|
| $A_1$ | `R0 = counter;` | $B_1$ | `R1 = counter;` |
| $A_2$ | `R0 = R0 + 1;` | $B_2$ | `R1 = R1 - 1;` |
| $A_3$ | `counter = R0;` | $B_3$ | `counter = R1;` |

Consider one possible ordering: $A_1$ $A_2$ $B_1$ $A_3$ $B_2$ $B_3$ , where `counter=17` before starting. Uh oh.

What we have here is another example of a race condition.

You may be thinking, "well, what are the chances, one in a million that the scheduler will choose to preempt the process at exactly the wrong time?"

Doing something millions or billions of times isn't really that unusual for a computer, so it would come up..

Concurrency can be quite challenging, and inattention to details may result in programs that don't work as expected. Some of the programs that we have had you write involve thread objects have been designed so that no interference is possible. But it is important to this about the possibilities of interference when you use threads. Advanced Computer Science courses, especially Operating Systems, study concurrency in much more detail, including other problems that may arise and the techniques to deal with them.