Computer Science 120
Introduction to Programming
Siena College
Spring 2011

Topic Notes: Active Objects

Repetition

People find repetition boring. Fortunately, computers don't feel this way. This is fortunate because repetition is the only way we can exploit the full power of a computer. As we discussed in the first class, part of the computer's power comes from the fact that it can follow the instructions stored within its memory rapidly without waiting for a human being to press a button or flip a switch.

In all of the examples we have considered so far, the sequences of instructions preformed when a mouse event occurs are quite short and then the computer has to wait for us again. The computer works for a fraction of a second then waits. We could get the computer to do more work in response to our mouse events by writing methods with thousands or millions of instructions, but this would be painful.

But we can get the computer to execute thousands or millions of instructions without writing thousands or millions of instructions ourselves: we can have the computer execute the same instructions over and over and over again.

At first, this may seem like a boring and inefficient use of the computer. In fact, when in comes to following instructions, doing the same thing over and over again can be very interesting. Think of the scribble program or the Spirograph program. Each time we drag the mouse in these programs, the computer "does the same thing" in the sense that it executes the same instructions — the body of `onMouseDrag`. Each time these instructions are executed, however, the computer actually does something different because the meaning of at least one of the variables referenced in the instructions, `point`, has changed.

Consider this example, where we get some "interesting" behavior through repeating the same instructions without depending on changes in the mouse position.

See Example: RailroadClick

Here, we draw a railroad track, one railroad tie at a time, by clicking the mouse.

Each time the mouse is clicked, the `onMouseClick` method does the same things. It creates a `FilledRect` that looks a bit like a railroad tie and it increases the value associated with the variable `tiePosition`. Because `tiePosition` is increased with each click, the next click draws its tie a little farther over in the screen. To prevent the program from wasting time by drawing ties no one will ever see, an `if` statement is included that skips the creation of new ties once `tiePosition` gets large enough.

It is painful to have to click repeatedly to get the ties drawn. Instead we would like the computer to continue drawing ties while they are still on the screen. Java provides the `while` statement, or "while loop", to perform repeated actions. Java includes other looping constructs that we will see later in the semester.

The syntax of a `while` statement is:

```
while (condition)
{
    ...
}
```

As in the `if` statement, the condition used in a while must be some expression that produces a `boolean` value. The statements between the open and closed curly brackets are known as the *body* of the loop.

A common way the while loop is used is as follows:

```
while (condition)
{
    do something
    change some variable so that next time you do
                    something a bit differently
 }
```

Armed with this construct, we can draw all of our railroad ties in the `begin` method.

See Example: Railroad

As in this example, the condition controlling the `while` loop will usually involve the variable that's changing. If nothing in the condition changes, then the loop will never terminate. Such a condition is called an *infinite loop*. We avoid this, in general, by ensuring that our loops have a precise stopping condition. While we might be able to look at an algorithm and say "hey, we should stop now", Java will not (and in fact cannot, in general) determine if a loop will not stop.

Active Objects

We have now seen how to get one set of commands to be executed repeatedly. But there certainly are programs that have different things happening simultaneously. That is our next goal.

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).

We can also create "active" objects in Java. They can contain instructions (in a special method called `run`) that run even when the user doesn't do anything with the mouse.

If you put a loop that goes on for a long time in an `onMouse...` method, the `WindowController` can't respond well to additional events because it is busy in the mouse-handling method. Instead, we will put such loops in the "`run`" methods of `ActiveObject`s.

To create an `ActiveObject` one must:

1. define a class that "`extends ActiveObject`"
2. define its constructor and say "`start()`" at the end.
3. define a "`public void run()`" method.

The class `ActiveObject` is part of the objectdraw library. It includes a number of instance variables and methods that are used to keep track of objects which can execute in parallel with each other. The method `start` does some housekeeping which results in the creation of a new "thread of control" (or just "thread"), which then begins running the `run` method. Thus evaluating `start` (or equivalently, `this.start()`), eventually leads to the creation of a thread that executes the `run` method. When the method terminates, the thread dies, and the object is no longer "active."

Our first example is of falling balls in a window.

See Example: FallingBalls

Consider the `ActiveObject` in this program - the `FallingBall`. We see that it does all the things we said an `ActiveObject` is supposed to do. Its class header tells Java that this class extends `ActiveObject`. We have several constants and an instance variable to hold the `FilledOval` that will be the actual ball.

The constructor includes the same functionality as things like the `Tshirt` examples - it constructs the objects that make up an instance of this class (in this case, just a `FilledOval`). The difference is that it must end with a `start()` statement.

The `run` method contains a simple while loop that defines the ongoing "activity" of a `FallingBall`.

The condition `ball.getY() < yMax` is true as long as the `ball` is on the screen. Notice that the body of the `while` loop contains the statement `pause(DELAY_TIME)`. When this is executed, the thread pauses execution for `DELAY_TIME` milliseconds (thousandths of a second) before going around to the next iteration of the loop to move the ball by another `Y_SPEED` pixels down the screen. If the `pause` statement were not in the loop the animation would take place so fast that we would not be able to see it. For each of our applications we will play with the value of `DELAY_TIME` until we get a speed of animation that looks the best. For this particular application, we chose a value of 33. If the value of `DELAY_TIME` were 66, the ball would fall half as fast (there would be twice as much delay between movements), while a value of 11 would have the ball falling 3 times as fast.

There is another, more technical, reason for including a `pause` statement in the loop. Many computers only have a single processor. Thus if two threads are active, they are both being run by the same processor. In order to make it look as through both are being run simultaneously, they take turns. Different computer operating systems have different ways of taking turns. Some automatically trade off after a certain time interval (usually every few milliseconds), while others wait for one thread to pause before releasing control to the other thread. We will always include pauses in every loop in the `run` method of an `ActiveObject` in order to ensure that they alternate turns fairly.

Once the ball has finished falling off the screen there is no need keep it around. We call `removeFromCanvas` to remove it. The `run` method then terminates, "deactivating" our `FallingBall`.

Notice that we can create more than one ball at a time. How does this work? Well, each time we click, a brand new instance of a `FallingBall` is created. That instance has its own copy of each instance variable: a `ball` and a `yMax`.

Different "Activities"

There is no rule stating that an `ActiveObject` must involve a constant motion of an object or objects. In fact, we have an incredible amount of flexibility in what can be controlled in an `ActiveObject`. Consider this example:

See Example: VanishingScribble

This program looks a lot like our scribble program from earlier. However, the objects that look like regular `Line`s that we draw in this one are really `ActiveObject`s that will cause the `Line`s to go away after a period of time.

When we press the mouse, we decide if we are going to draw the next scribble with `FallingLine` objects or `FadingLine` objects. Each is an extension of `ActiveObject` which will cause the line to disappear at some point.

First, let's look at the `FallingLine`. In many ways, it resembles the `FallingBall` object. We draw something on the canvas, activate it, and the `run` method moves it until it leaves the canvas, at which point we remove it.

The difference here is that we've added some acceleration. The `Line` object controlled by the `FallingLine` waits a bit, then slowly starts to move the `Line` down the screen. To simulate acceleration due to gravity, we have the speed increase by 10% each time we move the line a bit. Once both endpoints have left the canvas, we remove the `Line` and let this `ActiveObject` deactivate when the `run` method completes.

The `FadingLine` does something a little different. This `ActiveObject` extension doesn't move its `Line` at all: it just changes its color. We start out with a black line, wait a bit, then slowly change the color from black, through the greys, until it becomes white. At that point, we remove it from the canvas.

Building a "Pong" Game

Next, we'll build a pretty boring pong game (unlike the *real* Pong game which is incredibly exciting), since the paddle can't actually hit the ball, but it does demonstrate that the paddle can move at the same time that the ball falls.

See Example: PatheticPong

We build a playing area at the bottom of which we draw a paddle. This paddle will follow the x position of the mouse as is moves within the window, subject to the restriction that it always stays within the playing area. Each time the mouse is clicked, a new falling ball is created at the top of the playing area that starts falling.

Notice the extra work being done in `onMouseMove` to make sure we never allow the paddle to go outside the playing area. If the mouse's x position goes too far left or right, we draw the paddle at the legal position (within the playing area) closest to the mouse position.

How does Java manage both the paddle and the falling balls? There is always one thread that handles the mouse motion methods (*e.g.*, `onMouseMove`), and a new thread is created each time we create a ball. So this program can have multiple threads operating concurrently: one to move the paddle and one to move each ball currently on the canvas.

Our first improvement will be to add the interaction between the paddle and a ball. If the ball strikes the paddle (or vice versa), the ball should then be redirected. For simplicity, we'll just assume straight up and down motion of a ball for now.

The first question: what part of our program will detect the contact between the ball and the paddle? We might think it should be the paddle that finds out if it is in contact with the ball, then tells the ball to start moving in a different direction. Or that the ball knows where it is and could ask the paddle if it has come in contact.

Either way has the potential to work, but there are two factors that will lead us to choose the former:

1. We can have the ball check for contact with the paddle in its `run` method, which will be executing for as long as the ball is in existence. We can only have the paddle check for contact when it receives a mouse event. This would preclude us from detecting the case where the paddle is stationary and comes into contact with a ball.

2. There is one paddle through the life of the program and it is known at the time we create each ball. So the ball can be given information about the paddle when it is constructed. The paddle, on the other hand, would need to know about all balls that are created and check for contact with each. It would also need to know when a ball ceases to exist.

Therefore, we will pass information about the paddle to the ball's constructor. When the ball moves, it will check to see if it is in contact with the paddle.

See Example: LessPatheticPong

Here, the `WindowController` class is very similar. It only adds the new parameters required by the `SimplePongBall` constructor.

The `SimplePongBall` is the `FallingBall` that has been enhanced to change direction (to go up) when it comes in contact with the paddle, and to change direction (to go down) if it reaches the top of the playing area. It takes the paddle and the highest point of the playing area as additional constructor parameters and remembers these values for use in the `run` method.

The `run` method now needs to keep track of the speed, which may be positive or negative depending on which direction the ball is travelling. In addition to the motion, we check to see if the ball has reached the top of the playing area. If so, we make sure the ball is travelling downward (positive Y speed). We also check to see if the ball is in contact with the paddle using a new method called `overlaps`. If so, we make sure the ball is going to move upward (negative Y speed).

If the ball ever falls below the bottom of the canvas, it is removed and the `run` method returns, deactivating this ball.

Our final enhancement, at least for now, is to add horizontal motion to the ball, and to make it bounce off the side walls

See Example: Pong

Our `Pong` class is almost identical to `LessPatheticPong`. The only difference is in what we have to pass to our ball, now called a `PongBall`. The `PongBall` now needs to know about the boundaries of the playing area, which we can conveniently provide by passing the `FramedRect` we call `boundary`.

The main things we need to be concerned about in the constructor are determining the boundaries of the playing area and drawing the ball at its initial position. We determine the playing area bounds from the geometry of the `FramedRect` passed in, with a little extra work to account for the ball size for the right side (why?).

We then generate a random x position within the playing board, and create the ball at that position just below the top of the board. Notice a new way to generate a random number: the `RandomDoubleGenerator`. It works just like a `RandomIntGenerator` except that it generates (surprise!) `double`s instead of `int`s.

We don't get the ball moving until down in the `run` method. There, we first choose initial speeds in the x and y directions, again randomly. Then as we move the ball, we check to see if we've hit a wall or the paddle, adjust our speeds accordingly to take care of the bounce, and move the ball.

This is far from perfect, but somewhat playable. Probably the most obvious flaw is that we always simply reverse the y speed when the ball strikes the paddle, even if the ball hits the side instead of the top. That doesn't seem very natural. We also have no way for multiple balls to interact with each other – they simply pass through each other magically.

Talking to an `ActiveObject`

The `ActiveObject`s we have seen so far are created and then do their own thing (their `run` method) without any further instruction from the main class. However, we are not restricted in that way. We can send messages to `ActiveObject`s just as we can send them to other objects.

Consider this minor enhancement to our pong game:

See Example: TiltPong

This one will allow us to give a little "nudge" to the most recently created ball by clicking the mouse outside of the playing area. We do so by adding a method `nudge` to the `PongBall` class and calling that method at the appropriate time in our `WindowController`, which is now called `TiltPong`. To give us an indication that this worked, the `nudge` method will also temporarily change the color of the ball to red.

In order to have the speed variables affected by the `nudge` method, they have been changed from local variable of `run` to instance variables. When `nudge` is called, the `xSpeed` and `ySpeed` variables are changed so that the next time the `while` loop in the `run` method moves the ball, its speed will be different.

The only other change in the `PongBall` class is to change the ball's color back to black each time around the `while` loop in the `run` method.

The changes to the `TiltPong` class are simple: a new instance variable in which we remember the most recent ball, and a call to the ball's `nudge` method when the mouse is clicked outside the playing area.

Graphics Images (and more advanced `ActiveObjects`)

Our goal is this example:

See Example: FallingSnow

In order to achieve this, we first need to figure out how those snowflakes can be drawn.

Drawable Images

Consider this example:

See Example: Snowman

In the program above, we drag a picture of a snowman around the screen. The picture comes from a "gif" file named `snowman.gif`.

The image is certainly too complex to draw using our ObjectDraw primitives. Fortunately, we can read an image from a file and save it as an object with type `Image`. `Image` is a built-in Java class from the library `java.awt`. Hence you need to make sure that any program using `Image` imports `java.awt.Image` or `java.awt.*`.

The first line of the `begin` method of the `Snowman` class shows how to do this when given a "gif" file (a particular format for holding images on-line):

```
    snowManPic = getImage("snowman.gif");
```

where `snowManPic` is an instance variable declared to have type `Image`. Downloading a "gif" file can often be slow, so we usually will want to create an image from the "gif" file at the beginning of a program and save it until we need it. If you download "gif" files in the middle of your program, you may cause a long delay while the computer brings it in from a file on a local disk or fileserver.

While objects of class `Image` can hold a picture, they can't do much else. We would like to create an object that behaves like our other graphics objects (*e.g.*, `FramedRect`) so that it can be displayed and moved around on our canvas.

The class `VisibleImage` from the ObjectDraw library allows you to treat an image roughly as you would a rectangle. In fact, imagine a `VisibleImage` to be a rectangle with a picture embedded in it. You can do most things you can do with a rectangle, except that there's a neat picture on top.

To create a new `VisibleImage`:

```
    new VisibleImage( anImage, xLocation, yLocation, canvas);
```

For example, `new VisibleImage(snowManPic, 10, 10, canvas);` would create an object of type `VisibleImage` from the image in `snowManPic` and place it on `canvas` at location (10,10), with size equal to the size of the image it contains.

If you associate a name with your `VisibleImage`, you can manipulate it using some familiar methods:

```
    VisibleImage snowMan;
```

And then later:

```
    snowMan = new VisibleImage(snowManPic, 10, 10, canvas);

    snowMan.setWidth(124);
    snowMan.setHeight(144);
```

Our original snow man image is large: 619x718 pixels, but we shrunk him down to a more reasonable size.

What do you think happens if we say:

```
    snowMan.setColor(Color.green);
```

Nothing! It's not an error, but nothing is done for you either! Because the picture already has its own colors, it wouldn't make sense to change it to a solid color. Similarly, the value returned by `snowMan.getColor()` is always `Color.black`, no matter what colors are in the image!

The rest of the code for the `Snowman` class is just a variation on the earlier programs that allowed us to drag around squares and T-shirts.

Multiple `ActiveObject` types

We're now ready to consider the example:

See Example: FallingSnow

First consider class `Snow`, which extends `WindowController`. While it includes code for loading the images of the snowflakes and draws the background picture, the only indication that something interesting is going on is in the method `onMouseClick`. On each click, a new object of type `Cloud` is created. It is the `Cloud` object that is responsible for all those snowflakes. The snowflake image is passed as a parameter to the `Cloud` constructor.

First look back at the code for class `FallingBall`. A falling snowflake will be very similar, except that the object falling will be a `VisibleImage` rather than a `FilledOval`. But there's more to it than this.

We created a FallingBall every time the user clicked the mouse. When the user clicks the mouse now, the process of generating and dropping snowflakes begins. There's another class here: the Cloud, which we've also made an ActiveObject. A cloud is an ActiveObject that continuously generates snowflakes. Each snowflake is an ActiveObject that, when constructed, floats down the screen.

Let's jump right to the run method of the Cloud class. It starts by declaring a local variable, snowCount, initialized to 0. The rest of the method is a while loop which increments snowCount, constructs a FallingSnow (more on FallingSnow soon!), and then pauses for 900 milliseconds before repeating. The while loop's body will be run 150 times before stopping.

The constructor for Cloud saves its parameters as instance variables. Both will be needed for the calls to the FallingSnow constructor in the run method. Since the values of the formal parameters go away at the end of the method or constructor in which they are declared, we need to save them in instance variables.

The constructor also creates snowGen, a random number generator used to determine where each snowflake will be dropped and how fast it travels down the screen.

Finally, the constructor calls start(), as required to activate our ActiveObject.

Going back to method run, we will see below that the constructor for FallingSnow takes parameters which are a DrawingCanvas on which to draw the image, the Image of the snowflake, a double that determines how far from the left edge of the screen the snowflake will be located, another double to indicate how fast the snowflake falls, and finally an integer indicating the height of the window.

Two of the actual parameters: canvas and snowflakePic are values of instance variables that were provided values by the constructor of Cloud. Cloud simply remembers and passes along these values to FallingSnow and never uses them in any other way. The other two actual parameters are random values generated by snowGen, which provides values between 0 and the width of the screen. The next to last parameter, representing how far the snowflake falls each time through a loop, is calculated from such a random number, but is divided by the canvas height in order to reduce the speed to a smaller number. What values are possible for this parameter? It is computed by the line

```
snowGen.nextValue()*2/canvas.getWidth()+2
```

Of course we could have used a separate random integer generator that provides only smaller numbers in this range, but the above expression also works.

Let's now take a look at the FallingSnow class, another ActiveObject.

The constructor remembers the speed and canvas in instance variables so that they can be used later in the run method, and then creates a VisibleImage from the Image of a snowflake. Once the image has been embedded in a VisibleImage, we can move it around on the screen. In fact, since we created the image at the coordinates (0,0) – the upper left corner of the screen, we immediately move it to its correct x-coordinate and set its y-coordinate so that the bottom of the snowflake is off the top of the screen.

Why not just create the snowflake at the right position instead of creating it at (0,0) then moving it? Unfortunately, we cannot determine the dimensions of a `VisibleImage` until it has been created. That is, we cannot get this information from the `Image` used in constructing it. Thus we had to first construct the snowflake before we could see how high it was, and thus, how far up the screen it needed to be located so that it would not be seen! We could have created the `VisibleImage` with x-coordinate `x`, but since we knew we were going to have to move it anyway, we just created it at (0,0) and then moved it both across and up.

As usual, the last line in the constructor is the command `start()`.

The `run` method of `FallingSnow` is quite simple. It is a simple loop that pauses and then moves the snowflake. It terminates when the snowflake is off the screen. It then removes the snowflake from the canvas.

When executing, this program contains several passive objects and may contain hundreds of active objects, all running at once. There is an object corresponding to the main class, `Snow`, that loads the snowflake pictures and draws the scene. It responds to mouse clicks by creating an object of class `Cloud`. The creation of a `Cloud` results in the creation of 150 objects of type `FallingSnow`.

`ActiveObject` recap

To recap, to create an `ActiveObject` you:

1. define a class that “extends `ActiveObject`”
2. define its constructor and say `start()` at the end.
3. define at least a `public void run()` method.

To see why we include the `pause` method call in the `while` loop of `ActiveObject`’s, look at the behavior of a minor variant of the program where the only change is that we omit the `pause` in the `while` loop of class `Cloud`.

The difference is that all of the snowflakes are generated without pause, so they all essentially start at once (though some are slower to fall than others). The `pause` makes the animation much more obvious. (What would have happened if we omitted the pause in the `FallingSnow` class?)

More Advanced Loops

Now that we have seen how important loops are in `ActiveObject`s, we step back and discuss more complex loops.

See Example: Knitting

Here, each time the mouse is clicked, we knit a scarf.

If you look carefully at the pictures generated, you will see that the scarf is formed by overlapping circles. It is easiest to develop this by first writing code to generate a row, then expand it to generate the correct number of rows, in the correct positions.

To draw a row, we will have a `while` loop. Each time through the loop we increase the value of `x` position as well as bump up our counter of the number of columns drawn so far, `numCols`.

That wasn't too hard, but now we'd like to create successive rows. Each time we start a new row, there are a number of things that we will need to take care of:

1. We need to reset the value of `x` so that we start drawing at the beginning of the row rather than where we left off.
2. We to increase the value of `y` so that rows won't be drawn on top of each other.
3. We need to reset `numCols` back to 0 so that it will keep the correct count when we restart drawing a row.
4. We need to bump up `numRows` each time through.

Now all we need to do is to repeatedly execute the code for drawing a row by placing it inside an enclosing `while` loop. This is our first example of a *nested loop* structure: a loop that executes within a loop.

There is nothing mysterious about a nested loop. The inner loop is simply part of what the outer loop does over and over.

See Example: FlagMaker

Next, we will look at another program that uses nested loops. We will draw 48-star American flags.

Most of what we see in this program is familiar.

The main thing we use here that we have not seen previously are a couple of *private methods*.

The methods `drawStripes` and `drawStars` are called from inside the `Flag` constructor. They are not designed to be accessible from outside the class. They are designed only to be useful in breaking down the constructor into easier to understand pieces. `drawStripes`, is especially useful because it allows us to avoid duplicating code. Notice that it is used twice inside the constructor. Once to draw short stripes, and once to draw long stripes. Because we provide different parameters to it each time, it produces different results. If we did not use this private method, we would have to repeat the code in the method twice, once for each collection of values of the parameters.