Computer Science 252
Problem Solving with Java
The College of Saint Rose
Fall 2013

Topic Notes: Java and Objectdraw Basics

Event-Driven Programming in Java

A program expresses an algorithm in a form understandable by a computer.

That "understandable" form is a program and must be written in a *programming language*.

There are many, many programming languages, each of which has its own advantages and disadvantages. We teach our introductory sequence in one particular (and very popular) language: Java.

We choose Java because it is in wide use, can be used to write programs that perform a wide variety of tasks to run on a wide variety of computers. It is also *object oriented*, a term we will see in more detail soon.

We will see two main types of programs. Some of our programs will execute from beginning to end to compute a set of outputs (usually text printed to the screen or to files on the computer's disks) from a set of inputs (entered at the keyboard or read from disk files). These are what you are probably most familiar with from your CSC 202 class or other experience.

The majority of our programs will be *event-driven programs*. These are more interactive and, in our case, graphical. An event-driven program responds to actions such as a mouse click or a key press by performing some specific action, then waits for the next event.

Java was designed with events in mind, and we will take advantage of this. It means we can write programs that respond to mouse movements and clicks, and we will use those programs to display and manipulate some simple graphical objects.

A First Program

So we consider our first "real" event-driven Java program:

See Example: TouchyWindow

If we run the program, we see that it brings up an empty window. When I press the mouse button in the window, a message appears, and when I release the mouse button, it disappears.

While that in itself doesn't seem very exciting, keep in mind that the program we are running is very simple. It fits easily on one screen. Let's take a look at the text of this program and see what it all means and why this program does what it does.

```
import objectdraw.*;
```

```
import java.awt.*;
```

These `import` lines tell Java that our program is going to build upon some code that's already been written by others. "objectdraw" is a software library developed by the authors of our text that will allow us to write event-driven graphical programs without worrying about some of the gory details. "java.awt" is part of the standard Java library that helps to display windows on the screen.

These two lines will appear at the top of nearly every program we write this semester. Nearly all Java programs begin with a series of `import` lines to bring in the building blocks they will use.

You have almost certainly imported things like Java's `Scanner` and `Random` classes in previous programs.

```
/*
 * A first Java/objectdraw example.
 * From Bruce, Danyluk, Murtagh, 2007, Chapter 1.
 *
 * $Id: objectdraw.tex 2218 2013-10-18 14:06:39Z terescoj $
 */
```

This next segment is a *comment*. Everything here between the `/*` and the `*/` is ignored by the computer. It is there entirely for our benefit – the humans who need to write or understand the program.

```
public class TouchyWindow extends WindowController {
```

This line gives us (and Java) a lot of information. First, the term `public` is telling Java that the program is "public" – we can run it. We'll see alternatives to `public` in some contexts, but every one of our programs will start this way.

The word `class` tells Java that we are about to define a "class". The reason for the term will become more clear soon.

`TouchyWindow` is the name of our program (and the name of the `class` that defines the program.

`extends WindowController` means that this new `class` we're defining called `TouchyWindow` is going to build upon ("extend") another, already existing `class`, called a `WindowController`. Essentially we're saying that we'd like to use a `WindowContoller`, but we're extending it to have some new functionality above and beyond, and we're calling that new `class` `TouchyWindow`.

The `WindowController` `class` is defined by the objectdraw library. It is what puts the window (i.e., the white box) up on the screen. By itself, it never displays anything in the window. It's up to us, in our extension, to make use of that box to do something (slightly) more interesting.

Lastly, there is a "{" character, which tells us that the *class header* is complete and now we're ready to start to define the *class body*.

In our case, the class body contains two *methods*:

```
/* This method will execute when someone clicks on the window.
   It will result in a message being displayed.
*/
public void onMousePress(Location point) {
    new Text("I'm touched", 40, 50, canvas);
}

/* This method will execute when the mouse button is released.
   It will remove everything drawn in the window, which in this
   case can only be the text message displayed by the above.
*/
public void onMouseRelease(Location point) {
    canvas.clear();
}
```

These methods are where the actual instructions are given. Each method is preceded by a comment describing what it does. But we'll look at the methods themselves.

There are two methods defined: `onMousePress` and `onMouseRelease`. In each case, the name of the method is preceded by "`public void`" and followed by "`(Location point)`. For now, we'll just say that these methods need to have these extra words and symbols – their meanings and what else we might put in those positions will come later. This is all called the *method header*.

Following the method header, there is again a { character, which denotes the start of the *method body*.

In each of our methods, the method body consists of a single Java statement. In `onMousePress`, we tell Java that we want a `new` piece of `Text` to be drawn on our screen, and we specify what text we want, where it should be placed (40 and 50 are *coordinates* – more on this soon), and on what we should draw it (the `canvas`, which is objectdraw's name for the window placed on our screen by the `WindowController`).

Specifically, `Text` is a class, defined by the objectdraw library. When we say "`new Text`", we are instructing Java to find the `class` definition for `Text` and *construct* an *object* of that class. The specifics of how to create that `Text` object are determined by the *parameters* listed in parentheses after "`new Text`".

In the `onMouseRelease` method, the statement is an instruction to the `canvas` to erase anything that's been drawn on it.

Note that each method and the class definition itself is terminated by a "}" character. This ends the definition of either the method body or class body that was started by a { character.

So we have a complete program – why does it make our program do what it does when we run it?

As their names suggest, the instructions in the bodies of our methods execute in response to mouse events. Specifically, when someone presses the mouse button in our window, the `WindowController` looks for a method named `onMousePress` and executes the statements in that method. Similarly,

when the mouse is released, the instructions in `onMouseRelease` are executed.

You'll notice that there is no `main` method here – the program does not do anything (beyond the creation of the "canvas" which is handled by the `WindowController`) until we interact with it using the mouse.

Other Mouse Event Methods

As you might guess, there are other "mouse event" methods available that we can use to make our program more responsive. Any class that `extends WindowController` may define:

```
public void onMouseClick(Location point)
public void onMouseEnter(Location point)
public void onMouseExit(Location point)
public void onMousePress(Location point)
public void onMouseRelease(Location point)
public void onMouseMove(Location point)
public void onMouseDrag(Location point)
```

Finally, there is one additional method we can define in a `WindowController`, called `begin`. It looks very similar to the others except that it doesn't have the "`Location point`". The `begin` method, as its name suggests, executes exactly once: when the program begins.

We will soon make use of `begin` and more of the mouse event handlers, but first, we'll take a look at what else we can draw besides bits of text.

Graphics Primitives

To fully understand the instructions within the method bodies we have examined, you need to understand how the system for drawing graphics within a Java program work.

To place an object on the screen, you include an instruction called a construction in a method. Each construction will include:

- The word `new`
- The name of the type of thing you want to draw. Possibilities include:

  ```
  FramedRect, FilledRect
  FramedOval, FilledOval
  Text, Line
  ```

- a list of extra bits of information called *actual parameters* that determine the size and position of the object displayed.

Some examples:

```
new FramedRect(10, 10, 40, 60, canvas);
new Line(x1, y1, x2, y2, canvas);
new Text("hello there", x, y, canvas);
new FilledOval(100, 100, 30, 60, canvas);
```

The most important of the parameters included in these constructions are those that specify the locations and dimensions of objects. They are interpreted in a coordinate system in which:

- The basic unit of measurement is one dot on the computer's display (*i.e.*, one *pixel*).
- The y-coordinate is "upside down" compared to the convention from mathematics (*i.e.*, the bigger the y-coordinate, the closer to the bottom of the screen).
- The *origin* (*i.e.*, the point (0,0)) is located in the upper left corner of the program's window (not of the display).

For the `FramedRect`, this draws the outline of a rectangle with the upper left corner at (10, 10), with a width of 40 and a height of 60. So where is the lower right corner?

The `Line` is drawn from `(x1,y1)` to `(x2, y2)`.

The `Text` is drawn with its upper left corner at `(x, y)`.

The `FilledOval` is drawn within an "imaginary box" with its upper left corner at (100, 100), width of 30, height of 60.

Looking back at the TouchyWindow example, we can see that the text is in fact placed at coordinates (40,50) in this coordinate system.

Giving Names to Objects

Now, let's experiment a bit with these different event types and object types.

See Example: ColorEvents

There are two new things in this example. First, we need to know how to set the color of an object. This is done with the statement:

```
setColor(Color.xxx);
```

where "`xxx`" is one of the colors Java knows about.

But just saying "`setColor`" isn't enough – we need to tell Java what object's color is supposed to change.

To do this, we need to give the object a name. This is the other new thing in this example. These names are called *variables*.

In order to use a variable to give a name to an object, we need to do two things:

1. We must *declare* the variable. In this case, we are declaring *instance variables* since they are defined inside of our class, but outside any method body. We will see other types of variables later.

   ```
   private FilledOval oval;
   private FramedRect rect;
   private Line line;
   ```

   A declaration "introduces" the name to Java, so when we use it later on, it knows what the name "refers" to. In this case, we're saying that the name `oval` is going to refer to a `FilledOval` object.

2. We must associate a value with the variable. This is done using an instruction called an *assignment statement*.

   Our example has three assignment statements:

   ```
   oval = new FilledOval(50, 50, 100, 200, canvas);
   rect = new FramedRect(200, 10, 50, 100, canvas);
   line = new Line(20, 300, 300, 20, canvas);
   ```

   Note how we construct the object on the right hand side of the assignment operator (the `=`) and put the name where we wish to remember the object on the left.

Note that we can use any name we want for our variables. There's nothing saying we couldn't use the name "`oval`" for our `FramedRect` and "`rect`" for our `FilledOval`. But that would be confusing. It's always very good practice to use meaningful names (and we'll take points off your labs and projects if you don't). It makes the program easier to read and to understand.

There are a few restrictions on the words we can use with names:

- Names must start with a letter.
- Names are case sensitive.
- Letters, digits, and underscores may be used in names.
- Names may not be a word already used by Java (like `class` or `extends`).

Further, Java programmers generally agree upon a set of *naming conventions*. We will look at these in more detail as we go on, but for now, we will name all variables using lowercase letters, except when we have a name that is made up of multiple words, in which case we capitalize all but the first word. For example, if we want to give a name for a little red circle, an appropriate name would be `littleRedCircle`. Other variations such as `LittleRedCircle`, `LITTLE_RED_CIRCLE` or `LiTtLeReDcIrCle` would be valid names, but would not follow the naming convention for variable names.

Now that we have our variables and have assocated objects with them, we can use those variables to tell Java which objects to use for our `setColor()` statements.

```
rect.setColor(Color.blue);
```

Just like our mouse event handlers (*e.g.*, `onMousePress`) are methods of our `WindowContoller` classes, `setColor` is a method of the classes that define our graphics primitives (in this case, the `FramedRect`). The above shows how we call a method of a class.

A good way to think about this is that we are "sending a message" to the object. So we have the name of this `FramedRect`, and we're saying "hey `rect`, set your color to blue!".

We will soon see many more methods that will allow us to send messages to the graphics primitives, and we'll write our own methods for the more complex graphics objects we'll define ourselves.

This next example uses one more method to modify an object: the `move` method.

See Example: SunAndMoon

Everything here is familiar except:

```
heavenlyBody.move(0, 1.5);
```

As you might guess, this message tells the object named `heavenlyBody` to move 0 pixels in the x direction and 1.5 pixels in the y direction (down).

Every time we move the mouse in the window, this code executes, moving the sun down a bit. But in the `onMouseDrag` method, the circle moves by -1.5 in the y direction, so it moves up.

Accessing the Mouse Location

There is another important situation in which names are used to refer bits of information your program needs to work with. When the instructions within an event handling method such as `onMousePress` are followed, it is sometimes handy to refer to the coordinates where the mouse is located when the event occurs. Java makes this possible by letting you give it a name that should be associated with this information within the header of the method.

In fact, Java doesn't just let you provide such a name — it requires that you provide one. That is why we have had to include the text "`(Location point)`" in the header of each mouse event handling method we have written. This phrase tells Java that we want to be able to use the name `point` to refer to the place where the mouse is located. We just haven't actually used this ability yet.

See Example: MouseDroppings

This program places a small red circle on the canvas every time the mouse pointer moves.

The only line of interest here is

```
new FilledOval(point, 10, 10, canvas).setColor(Color.red);
```

Two things are different here from previous examples.

First, we have replaced the first two parameters to the `FilledOval` construction, which specify the x and y coordinates of the oval, with a single parameter, "`point`".

Each time the mouse is moved, before following the instructions in our method body, Java makes the name "`point`" refer to the coordinates of the current mouse position. When it sees the name `point` in the construction, it uses the coordinates of the mouse as if we had typed them in while writing the program.

When used in this way, the name `point` is called a *formal parameter*.

Note that the phrase "`Location point`" looks a lot like a variable declaration. The name "`Location`" describes the kind of thing that `point` will refer to just as the "`rect`" in

```
private FramedRect rect;
```

described the kind of information that could be associated with the name `rect`.

There is nothing special about the word "`point`" in this situation other than it appears in the method's header. Just as we can choose any word we want to use for an instance variable name, we can choose things other than "`point`" as a formal parameter name. If we take the method from this example and replace all the "`point`"s with a different name like "`mouseLocation`", the program will work the same way.

Remembering information between events

Now that we have seen how to use the mouse location for an event, let's consider a case where we need not only the **current** mouse location, but a **previous** mouse location as well.

We will construct a program to draw "Spirographs" – when the mouse is pressed then dragged, a series of lines are drawn from the press point to the current location.

See Example: Spirograph

Note that in this example, the only thing done in `onMousePress` is to save the value of the formal parameter `point` in an instance variable `linesStart`. If we did not do this, the value of `point` would be lost.

The instance variable declaration is of type `Location`. That makes sense – `point` is a `Location`, so the instance variable we'd use to store its value would also be a `Location`.

Then in the `onMouseDrag` method, we use the saved `Location` in `linesStart` as one end-point of a `Line` that we draw to the current `point` from `onMouseDrag`'s formal parameter.

Now let's consider a small variation – in `onMouseDrag`, rather than simply drawing a `Line`, we'll also update the saved `Location` value.

What have we done? We've created a "scribber" drawing program!

See Example: Scribble

And now, we'll look at an example where we create an object in response to one event and change it in response to subsequent events.

See Example: RubberBand

Here, we start by drawing a very small line – from the `pressedPoint` to itself – when the mouse is pressed. We remember that `Line` in an instance variable.

Then when the mouse is dragged, we modify that `Line` to have a new endpoint at the current mouse location. The result is a "rubber banding" effect.