Computer Science 252
Problem Solving with Java
The College of Saint Rose
Spring 2016

Topic Notes: Java Review 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, both academically and industrially, 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). Some of you have seen both types of programs, while others are most familiar with applications from your CSC 202 class or other experience.

Our first concern will be to introduce or refresh your memory about *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 Simple Program

So we consider this "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 2848 2016-01-25 03:52:30Z terescoj $
 */
```

This next segment is a *comment*. As you know, 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 include a class that starts with "`public class`".

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)
```

By adding any of these to our `WindowController` class, we are "teaching" our program what it should do when that particular mouse-related event occurs. If a particular event handler is not provided, it means we do not care about that kind of mouse event, and our program will ignore occurrences of those mouse events.

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.

Recall that 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 "Spirals of Lines" – when the mouse is pressed then dragged, a series of lines are drawn from the press point to the current location.

See Example: SpiralLines

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 endpoint 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.

Using Numbers

So far we have been using numbers only as coordinates for graphical objects. We'll soon be using them for much more. We'll first look at a very simple example that counts the number of times the mouse has been clicked in the window, and displays a `Text` object showing the current count.

See Example: ClickCounter

One of our instance variables, `count`, is of type `int` (for "integer"). This is a variable that gives a name to a number (rather than to a graphical object). It can contain any integral value from about negative 2 billion to positive 2 billion. Certainly plenty for our purposes!

In the `begin` method, we give this variable its initial value of 0. Then in `onMouseClick`, we add one to its value and reassign that result back to `count`.

This example is also the first one that demonstrates an important feature of good programming style: the use of *named constants*. Note the following lines at the top of the class body, just above our instance variable:

```
private static final int DISPLAY_X = 150;
private static final int DISPLAY_Y = 200;
```

As our programs become more complex, we will be using many numeric values. Using many somewhat arbitrary numeric values in a program can make the program difficult to understand and modify. We can improve the situation by associating the values with names so that we are reminded what the values signify when we see the names used.

Java a mechanism to enable us to use such names effectively. If you include the words "`static final`" in a variable's declaration, this indicates that the value assigned to it in the declaration will never change. The most important word here is "`final`". This means that its value cannot be changed (possibly by mistake)

Note that not everything can be a constant. Constants may not depend on anything created when the program starts up (except other constants). In particular, a constant may not depend on `canvas`. Thus we may never have a constant of type `FilledRect`, for example.

Conditional Execution

We have considered examples where we needed to remember a point (*i.e.*, a `Location`) or a graphics object (*e.g.*, a `Line`) from one event to the next. Now, let's look at an example where we need to remember both a point and a graphics object from one event to the next. This program draws rectangles interactively. When the mouse is pressed, the coordinates are saved as one corner of a rectangle to be drawn. As the mouse is dragged, a rectangle is drawn with that point as one corner, the current point as the other. Finally, when the mouse is relased, the final rectangle is drawn.

Most of what we need to do here is similar to previous examples. But let's think through how to approach this.

First, which event handlers will we need? We need to start drawing when the mouse is pressed, need to redraw the rectangle as the mouse is dragged ("temporary" rectangles that provide visual feedback to the user), and draw our final (permanent) rectangle when the mouse is released.

Next, we will consider what information we need in each event handler and how we can get our hands on that information.

When the `onMousePress` method is invoked, the parameter will give us the coordinates of one corner of the rectangles we'll be drawing. We don't draw anything yet, but this is information we will need. So we save it in an instance variable.

When the mouse is dragged, we receive the other bit of information needed to draw the temporary rectangle: the current mouse position. We can use another form of the `FramedRect` constructor, one that takes two `Location`s, to draw the appropriate temporary rectangle.

```
new FramedRect(firstCorner, point, canvas);
```

But there's more. What if this wasn't the first mouse drag event? Then, we need to remove the previous temporary rectangle before drawing a new one. That means we had better give a name to the temporary rectangle in an instance variable. Then, we can remove the previous temporary rectangle from the canvas before we draw the new one.

```
tempRect.removeFromCanvas();
tempRect = new FramedRect(firstCorner, point, canvas);
```

If we do this, we'll encounter some problems. The first time the `onMouseDrag` event handler is called, we will try to remove the object referred to by the name `tempRect`, but that name had never been assigned a value! Essentially, we are trying to send a message to nothing, and Java will respond by printing a long and messy error message. The first part of the message will mention a `NullPointerException`. That's usually a good indicator that you've made use of a name before giving that name a value.

So what do we need to do? We need to make sure we only try to remove the rectangle referred to by `tempRect` from the canvas if it has been drawn.

We'll use two steps to handle this. First, we'll give `tempRect` a special value in the `onMousePress` event to indicate that it does not refer to any object yet. In Java, an object name that refers to nothing can be assigned the special value `null`.

Then in `onMouseDrag`, we will first check to see if the value of `tempRect` is something other than `null`. If it is, that means it refers to an actual rectangle and we can remove it from the canvas.

We need to ask that question "is the value of `tempRect` not equal to `null`", which, as you know, is done in Java using a *conditional*, often referred to as an "if statement":

```
if (tempRect != null) tempRect.removeFromCanvas();
tempRect = new FramedRect(firstCorner, point, canvas);
```

We can then complete the program by implementing the `onMouseRelease` method. Here, we'll need to remove the last temporary rectangle from the canvas (but only if we know it exists!), then draw the permanent rectangle using the release point.

See Example: Rectangles

Clicking on Graphical Objects

Armed with the conditional construct, we can add a bit more meaningful interaction to our programs. We can react differently if a mouse event's location indicates that the mouse is over a particular object.

See Example: NudgeBall

In this example, we have a ball on the canvas. When the mouse is clicked, we move the ball to the right if the click point was inside the ball. The key line here is a conditional:

```
if (ball.contains(point))
    ball.move(BALL_MOVE, 0);
```

The `contains` method exists for all of our graphical object and tells us whether the given `Location` lies within the bounds of that graphical object.

We have now seen two types of *conditions* used as the test for our conditional statement:

```
if (tempRect != null)
```

and

```
if (ball.contains(point))
```

In both cases, we need to decide whether or not to execute the statement immediately following the `if` statement. This is done by evaluating the *boolean expression* inside the parens following the keyword `if`.

A boolean expression is one that must evaluation to either `true` or `false`. If it evaluates to `true`, the statement is executed. If it evaluates to `false`, the statement is skipped.

The first is the result of a comparison between a name and the special value `null`. We will see other comparisons soon.

The second is the result of sending a message to an object. This message returns either `true` or `false` depending on whether the object contains the given point.

This example also includes more constants.

```
// a constant defining the size of the ball
private static final int BALL_DIAMETER = 50;
// a constant defining the initial location of the ball
private static final Location BALL_POSITION = new Location(100, 100);
// a constant defining how far to move the ball when clicked
private static final int BALL_MOVE = 10;
```

In addition to `int` constants like we have seen previously, this example also has a `Location` constant.

Random Numbers

For our next example, we will introduce some randomness into our programs. Specifically, we will augment the "SpiralLines" example to pick a random color (from a set of 4 possible colors) for each spiral we draw.

We begin with the code from the original SpiralLines. If we would like each spiral we draw to have a randomly-chosen color, we need to make some enhancements:

1. We need to be able to choose a color randomly.
2. We need to be able to remember which color we chose so all of the lines in the spiral are drawn with that color.

Recall that Java's `Random` class provides this capability.

We'll want to choose a random integer from 0-3 every time the mouse is pressed and a new spiral is started. So in `onMousePress`, we pick a number and store it in another instance variable we'll call `colorNumber`.

But a number isn't a color; we need to translate that number into one of four colors to use. Once we've picked a color, we will store it in an instance variable named `currentColor`, declared as type `Color`.

```
if (colorNumber == 1) {
     currentColor = Color.red;
 } else if (colorNumber == 2) {
     currentColor = Color.blue;
 } else if (colorNumber == 3) {
     currentColor = Color.magenta;
 } else {
     currentColor = Color.green;
 }
```

Now, we have assigned the name `currentColor` to be one of four colors based on a randomly-chosen value. All that remains is to apply that color to each line drawn in the `onMouseDrag` method.

See Example: ColorfulSpiralLines

Using Custom Colors

So far we have used only a handful of pre-defined colors like `Color.red`, `Color.black`, *etc.*. We can extend beyond this limited color choice by creating our own objects of the class `Color`.

We can create any hue we wish by mixing the appropriate amounts of the primary colors of light: red, green, and blue. Computer monitors (and televisions, etc.) are typically made of lots of red, green, and blue light sources.

If we want a purple color to use for our graphics objects, we can mix red and blue:

```
purple = new Color(255, 0, 255);
```

The three parameters to the `Color` constructor are the amount of red, green, and blue to use. Each is in the range of 0-255, where 0 means don't use any of that color, 255 means use the maximum amount.

We will talk more later about creating just the color you have in mind. For now, let's think about how we can create an entirely random color.

Well, if a `Color` object is constructed from three numbers in the range 0-255, we can just generate three random numbers in that range and use them to construct our color.

See Example: MoreColorfulSpiralLines

In this case, we choose a color when the mouse is pressed and continue to use that same color for all the lines in a given spiral. We can make our program even more colorful by choosing a random color each time the mouse is dragged, thereby making each `Line` in our spiral a different color.

See Example: CrazyColorfulSpiralLines

Dragging Objects

A very common operation in our graphical programming will involve dragging items around the screen.

Recall that a "drag" involves pressing the mouse on the object to be dragged, dragging the mouse (with the button down) and having that object follow the mouse pointer, and finally, "dropping" the object at the position where the mouse is released.

How might we accomplish this?

1. We need to determine if the mouse is pointing at the object when it is pressed.
2. We need to move the object to follow the mouse while it is dragged.
3. We need to place the object at its new position when the mouse is released.

We'll start with a simple example that will allow us to drag a circle around the window:

See Example: UglyDragABall

Consider the three methods involved in performing the drag operation:

```
public void onMousePress(Location point) {

    if (ball.contains(point)) {
        // note that we've grabbed the ball and remember this point
        ballGrabbed = true;
    }
}
```

In `onMousePress`, we simply check to see if the location of the mouse press is inside the object we would like to drag. If so, we set a `boolean` instance variable to `true`.

A `boolean` variable is one that can contain only two possible values: `true` or `false`.

```
// update lastMouse location
public void onMouseDrag(Location point) {

    if (ballGrabbed) {
        ball.moveTo(point);
    }
}
```

While the mouse is being dragged, we check to see if the boolean variable is `true`. If so, we move the object to follow the mouse.

This also demonstrates a method of our graphics objects we have not yet seen: the `moveTo` method. This will take a graphics object that is already on the canvas and move its upper left corner to the given `Location`.

```
public void onMouseRelease(Location point) {

    if (ballGrabbed) {
        ball.moveTo(point);
        ballGrabbed = false;
    }
}
```

And now, when the mouse is released, we again check to see if the boolean is `true`. If so, it means we have been dragging and now need to move the object to its final position, and set the boolean back to `false` so we do not attempt to continue dragging this object (at least until the next time the mouse is pressed on the object).

But this is not a very satisfying "drag" effect. No matter where on the object the mouse is pressed, the object winds up having its upper left corner follow the mouse pointer. So when the object first starts to move, it appears to "jump".

Fortunately, this is not very difficult to fix. Consider this improved version:

See Example: DragABall

We add one more instance variable related to the dragging called `lastMouse` that remembers the most recent mouse `Location` for the `onMousePress` or `onMouseDrag` event.

What does this do for us? Well, if we move the object to be dragged by the **difference** between where the mouse **was** and where the mouse **is**, that object will move by exactly the same amount as the mouse just moved. This is precisely what we need to achieve a more natural "drag" functionality.

Rather than a `moveTo` in the `onMouseDrag` and `onMouseRelease`, we use a `move`:

```
ball.move(point.getX() - lastMouse.getX(),
          point.getY() - lastMouse.getY());
```

Note that we need to retrieve the `x` and `y` values from the `Location` values `point` and `lastMouse`, so we can compute the difference in each direction.

The `move` and `moveTo` methods can each be used to move objects on the canvas. `move` takes into account the current location of the object and moves an amount *relative* to that current location. `moveTo` does not depend on the current location but instead it moves the object to an *absolute* location.

More Complex Dragging

Let's use this idea of dragging to implement our most interesting program so far: one that plays a simple form of basketball.

See Example: Basketball

This example has many of the same constructs we have seen previously. The main thing we needed to add was a check to see if the ball was in the hoop if it was being dragged and the mouse is released. Only then does the player get credit for a basket.

Centering Objects

But before we move on, let's make one minor improvement: we will make the program work for different sized canvas settings, and we will set a larger font size and make sure our scoreboard text is centered horizontally on the screen.

See Example: Basketball2

In order to accomplish this, we will first need to see how to determine, in our program, the size of the canvas. Fortunately, this is readily available from objectdraw with the methods:

```
canvas.getHeight();
canvas.getWidth();
```

With this information, we can easily find important points, such as the center of the canvas, or points a certain percentage of the way down the canvas.

We will use these to place the objects on the screen. This means we can remove or replace some of our named constants. The constants that remain will indicate the percentage of the way down from the top of the canvas where we would like to draw the hoop, the scoreboard, and the ball.

Now, when we create each item that makes up the court, we will need to compute its position. We'll put the scoreboard aside for a moment and consider first the hoop and the ball. For each of these, we would like to draw them centered at a point half way across the canvas, and at a specified fraction of the way from the top of the canvas.

However, our object constructors do not specify the **center** of an object, but rather the **upper left corner**. This will complicate our calculation just a bit. We need to find the center, then subtract half of the width of the object to find the x-coordinate, and subtract half of the height of the object to find the y-coordinate.

Next, we consider the scoreboard. The first enhancement we'll consider is how to set the size of the font:

```
scoreboard.setFontSize(DISPLAY_SIZE);
```

But let's think about how we set the position correctly to center our scoreboard. Each time the text on the scoreboard changes, the width of the `Text` object changes, so we need to retrieve that width and use it to recenter our object each time the text changes.

We've started to use some more complex mathematical expressions. Consider this one:

```
canvas.getHeight() * BALL_FROM_TOP - BALL_SIZE/2
```

We have three *arithmetic operations* here, one multiplication (the `*`), one subtraction, and one division (the `/`).

Recall that Java proceeds based on a predetermined *order of operations*. The rule here is simple: the multiplicative operations (`*`, `/`, and `%` which is the modulo operator – used to compute a remainder) are performed first, from left to right. Then the additive operations are performed, from left to right.

So in the above, Java will first multiply `canvas.getHeight()` by `BALL_FROM_TOP`, then divide `BALL_SIZE` by 2, and then subtract the second result from the first.

We can override Java's default order of operations by parenthesizing subexpressions. For example, if we wanted to rewrite the expression

```
canvas.getWidth()/2 - scoreboard.getWidth()/2
```

To avoid dividing by two twice (essentially factoring out the division by 2), we would have to write

```
(canvas.getWidth() - scoreboard.getWidth())/2
```

Slightly More Realistic Basketball

We next update the basketball program to be a little more interesting of a game. Instead of dragging the ball to the hoop, we press where we want to take our shot from, drag in the direction we want to shoot, then release to take our shot. The ball is then moved to a point determined by the difference between the orginal press point and the release point.

See Example: BasketballShots

Doing Math with Colors

Our next example is a simple one, but demonstrates how we can create custom colors using arithmetic operations.

See Example: ColorfulSunset

Much of the example is similar to previous ones. We'll just focus on how the color of the sun changes as it sets.

Each time the mouse moves, in addition to moving the sun down by 1, we reduce the amount of green used to create the `Color` of the sun. The color starts out as a bright yellow: red=255, green=255, blue=0. Then as the sun sets, the intensity of green is reduced, leading to a smooth change as the color darkens through shades of orange before becoming red.

Of note here is that the example demonstrates a danger of using arithmetic operations to compute components of a `Color`: those values **must** be in the range 0–255 or Java will generate an error. Once `greenAmount` becomes negative, the construction of the new `Color` will fail and error messages are generated.

The example has a comment showing how we can add a conditional to fix this problem in this case.