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

Topic Notes: Conditionals and Numbers

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 *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 "Spirograph" example to pick a random color (from a set of 4 possible colors) for each Spirograph we draw.

We begin with the code from the original Spirograph. If we would like each Spirograph 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 spirograph 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 spirograph 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: ColorfulSpirograph

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: MoreColorfulSpirograph

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 spirograph. 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 spirograph a different color.

See Example: CrazyColorfulSpirograph

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
```

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.