Programming via Java User interaction

We'll now consider programs to get information from the user via the mouse, as supported by the `acm.graphics` package. Along the way, we'll learn about two more general Java concepts that will be useful in the future: **`null`** references and instance variables.

13.1. Accepting mouse events

To permit a program to allow a user interacts with the program via a mouse, the `GraphicsProgram` class defines several methods.

`void` `addMouseListeners()`

> Tells this program to receive information about the mouse's behavior and by invoking the mouse methods as appropriate.

`void` `mousePressed(MouseEvent event)`

> Invoked when the user presses the mouse button while the cursor is within this window. The parameter `event` contains information about the mouse click.

`void` `mouseDragged(MouseEvent event)`

> Invoked when the user drags the mouse — that is, the user moves the mouse while pressing the mouse button down. The method will be invoked even if the cursor has exited this window, as long as the initial mouse button press occurred within this window. The parameter `event` contains information about the mouse's state.

There are also `mouseReleased`, `mouseClicked`, and `mouseMoved` methods.

These mouse methods have a parameter, which is an object of the `MouseEvent` class, defined in the `java.awt.event` package. This `MouseEvent` object contains information about the mouse; among the `MouseEvent` methods, the most important by far provide information about the mouse's position at the time of the event.

`int` `getX()`

> Returns the *x*-coordinate where this mouse event occurred.

`int` `getY()`

Returns the *y*-coordinate where this mouse event occurred.

Figure 13.1 contains a simple program that allows some mouse interaction. The window initially appears blank; but when the user clicks the mouse within the window, a new circles appears at the click's location.

Figure 13.1: The `DropCircles` program.

```java
import java.awt.event.*; // for MouseEvent
import java.awt.*;       // for Color
import acm.graphics.*;   // for GOval
import acm.program.*;    // for GraphicsProgram

public class DropCircles extends GraphicsProgram {
    public void run() {
        addMouseListeners();
    }

    public void mousePressed(MouseEvent event) {
        double mouseX = event.getX();
        double mouseY = event.getY();
        GOval circle = new GOval(mouseX - 10, mouseY - 10, 20, 20);
        circle.setFillColor(new Color(255, 0, 0));
        circle.setFilled(true);
        add(circle);
    }
}
```

As you can see, the `run` method leaves the window empty, but it invokes `addMouseListeners` so that `mousePressed` will be invoked whenever the user clicks the mouse. When the user does click the mouse, the `mousePressed` method determines the mouse's *x*- and *y*-coordinates, and it places a solid red circle centered at that spot.

13.2. The `null` reference

Now we will try to change `DropCircles` so that no new circle is added when the user clicks within an existing circle. In order to do this, though, we need to need to study a new concept that we haven't seen before: the **`null`** reference.

In any place within a program where a reference to an object can appear, the **`null`** value can be used instead to indicate instead the absence of any object. For example, if we have a `GOval` variable named `ball`, and for the moment we don't want it to refer to any `GOval` object at all, we can assign it to be **`null`**.

```
ball = null;
```

Later in the program, we may ask the `GOval` named by `ball` to perform a method, such as move.

```
ball.move(3, 4);
```

The compiler will merrily accept this, and the computer will start to execute the program. If `ball` is **`null`** when the computer reaches this statement, though, the computer will complain that it is supposed to tell the `GOval` object to `move`, but in fact `ball` doesn't refer to a `GOval` object at all. It calls this a `NullPointerException`.

```
Exception in thread "main" java.lang.NullPointerException
```

A program should never request that **`null`** perform any methods, since **`null`** is the non-object.

You might wonder why it's useful to have a **`null`** reference at all. The `getElementAt` method in `GraphicsProgram` illustrates such a situation.

```
GObject getElementAt(double x, double y)
```

> Returns the topmost graphical object containing the point (`x`, `y`) in this window. If no objects contain that point, the method return **`null`**.

The designers of the `GraphicsProgram` class decided that the `getElementAt` method would be a useful method to have, but they had to confront this question: How should the method work in the situation where no objects contain the query point? They needed some way of returning nothing — and Java's **`null`** reference is perfect for this.

As you might guess, the `getElementAt` method is quite relevant to our goal of modifying the `DropCircles` program so that it adds a new circle only when the user clicks outside the existing circles. Figure 13.2 contains such a program, with only a few lines changed from before.

Figure 13.2: The `DropDisjoint` program.

```
1  import java.awt.event.*;
2  import java.awt.*;
3  import acm.graphics.*;
4  import acm.program.*;
5
6  public class DropDisjoint extends GraphicsProgram {
7      public void run() {
8          addMouseListeners();
9      }
10
11     public void mousePressed(MouseEvent event) {
12         double mouseX = event.getX();
13         double mouseY = event.getY();
14         GObject cur = getElementAt(mouseX, mouseY);
15         if(cur == null) {
16             GOval circle = new GOval(mouseX - 10, mouseY - 10, 20, 20);
17             circle.setFillColor(new Color(255, 0, 0));
18             circle.setFilled(true);
19             add(circle);
20         }
21     }
22 }
```

As you can see, its `mousePressed` method now invokes `getElementAt` (line 14) to see what object, if any, is at the point clicked by the user. If the method returns **`null`**, then the program knows there is no object at that point, and so within the **`if`** statement's body the program proceeds to add a circle into the window as before.

13.3. Instance variables

Now let us examine Figure 13.3, which allows the user to both to create new circles by clicking where no circles exist already and also to drag existing circles around the window.

Figure 13.3: The `DragCircles` program.

```
import java.awt.event.*;
import java.awt.*;
import acm.graphics.*;
import acm.program.*;

```

```
 6 public class DragCircles extends GraphicsProgram {
 7     private GObject cur;
 8 
 9     public void run() {
10         addMouseListeners();
11     }
12 
13     public void mousePressed(MouseEvent event) {
14         int mouseX = event.getX();
15         int mouseY = event.getY();
16         cur = getElementAt(mouseX, mouseY);
17         if(cur == null) {
18             GOval circle = new GOval(mouseX - 10, mouseY - 10, 20, 20);
19             circle.setFillColor(new Color(255, 0, 0));
20             circle.setFilled(true);
21             add(circle);
22             cur = circle;
23         }
24     }
25 
26     public void mouseDragged(MouseEvent event) {
27         double mouseX = event.getX();
28         double mouseY = event.getY();
29         double curX = cur.getX() + cur.getWidth() / 2;
30         double curY = cur.getY() + cur.getHeight() / 2;
31         cur.move(mouseX - curX, mouseY - curY);
32     }
33 }
```

This program uses an important new concept called an instance variable, declared in line 7. So far all of our variables have been declared *inside* methods, like `mouseX`on line 14. These variables, called local variables, exist only for the duration of the method; after the method completes, any value associated with the local variable is lost. By contrast, when a variable is declared *outside* all methods, like `cur` on line 7, that variable is an *instance variable*, and the value for an instance variable's value persists long-term, between method invocations.

You may be wondering: Why the word ***`private`***? This word does *not* make `cur` an instance variable: It is an instance variable because it is defined outside all methods. In fact, we could omit the word **`private`**, and `cur` would still be an instance variable, and the program would work just as before. Nonetheless, for stylistic reasons, instance variables should always be declared **`private`**. The compiler will prevent local variables from being declared **`private`**. We'll study the notion of privatenessmore in Section 14.3.

When the user presses the mouse button, the computer will execute `mousePressed`, and in line 16, the method will assign `cur` to reference the circle the user has selected. If the user clicks outside any existing circles, `getElementAt` will return **`null`**, and `mousePressed`'s **`if`** statement will execute. The **`if`** statement will create and add a new circle and assign `cur` to that.

Because `cur` is an instance variable, its value will persist even after `mousePressed` completes. Thus, when the user drags the mouse with the button still pressed, and the computer invokes the `mouseDragged` method, `cur` will still have the value it was assigned in `mousePressed`. The only job of `mouseDragged` is to reposition the circle referenced by `cur` at the mouse's current location.

When the user releases the mouse button, the computer will stop invoking `mouseDragged` as the mouse moves, since `mouseDragged` applies only while the mouse button is down. Later, when the user presses the mouse button again, the `mousePressed` method will assign `cur` to refer to the circle then selected (or the new circle thus created), so that the later `mouseDragged` invocations will affect that circle instead.

By now, you may be in the habit of always declaring all variables in each method the first time that they appear, so that you may be inclined to change line 16 to declare `cur` as a `GOval`. That is, you may be tempted to write `GOval cur = getElementAt(mouseX, mouseY);`.

Resist that temptation. If we write line 16 this way, then we would be defining a separate local variable named `cur` which hides the instance variable of the same name. The program would still compile, but `mousePressed` would work with the local variable `cur` rather than the instance variable `cur`. As a result, the program would never alter the instance variable `cur` from its default value, which is **`null`**. Consequently, `mouseDragged` would attempt to tell **`null`** to perform the`getX` method, which would lead to a program failure via a `NullPointerException`.

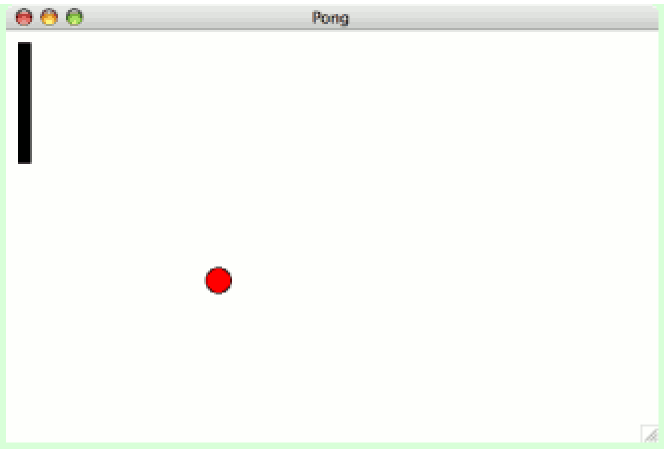
You may be tempted to declare *all* variables as instance variables, so that you don't have to declare any variable names multiple times. This is dangerous, because methods can end up interacting in peculiar ways. For instance, one method may invoke another, and they both happen to use the same variable name, but there is no intention of the methods interacting in this way. The program will fail.

Not only is declaring instance variables needlessly dangerous, but just as importantly it is very poor programming style. Only use instance variables where the program genuinely needs to remember their value long-term.

13.4. Another example: Pong

Let's look at a completely different example: Solitaire Pong. This is a simple game where a ball bounces around the window, except it will exit the window when it reaches the window's left edge. The user's mouse controls the vertical position of a paddle near the window's left edge. The goal of the game is for the user to keep the ball within the window as long as possible.

Figure 13.4: Running Solitaire Pong.



To accomplish this, we'll need a `run` method and a `mouseMoved` method. The `run` method will add the paddle and ball to the window and manage the ball's movement similar to the `BouncingBall` program (Figure 7.2). The `mouseMoved` method will read the mouse's *y*-coordinate and move the paddle to that location. Because both `run`and `mouseMoved` will need work with the same paddle, we will need an instance variable to refer to that paddle.

Figure 13.5 contains an implementation of these ideas.

Figure 13.5: The `Pong` program.

```java
import acm.program.*;
import acm.graphics.*;
import java.awt.*;
import java.awt.event.*;

```

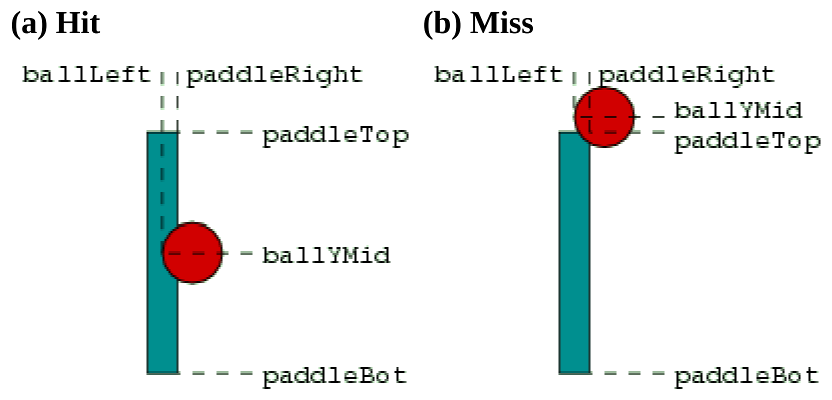
```java
public class Pong extends GraphicsProgram {
    private GRect paddle;

    public void run() {
        paddle = new GRect(10, 10, 10, 100);
        paddle.setFilled(true);
        add(paddle);
        addMouseListeners();

        GOval ball = new GOval(25, 25, 20, 20);
        ball.setFilled(true);
        ball.setFillColor(new Color(255, 0, 0));
        add(ball);

        double dx = 3;
        double dy = -4;
        while(true) {
            pause(25);
            double ballLeft = ball.getX();
            double ballYMid = ball.getY() + ball.getHeight() / 2;
            double paddleRight = paddle.getX() + paddle.getWidth();
            double paddleTop = paddle.getY();
            double paddleBot = paddle.getY() + paddle.getHeight();
            if(ballLeft + ball.getWidth() >= getWidth()
                    || (ballLeft <= paddleRight && ballLeft > paddleRight + dx
                        && ballYMid >= paddleTop && ballYMid < paddleBot)) {
                dx = -dx;
            }
            if(ball.getY() + ball.getHeight() >= getHeight()
                    || ball.getY() <= 0.0) {
                dy = -dy;
            }
            ball.move(dx, dy);
        }
    }

    public void mouseMoved(MouseEvent event) {
        double paddleY = event.getY() - paddle.getHeight() / 2;
        double paddleMax = getHeight() - paddle.getHeight();
        if(paddleY < 0) paddleY = 0;
        if(paddleY > paddleMax) paddleY = paddleMax;
        paddle.setLocation(10, paddleY);
    }
}
```

One of the most complicated portions of the program is the first `if` condition in `run`, for determining whether the ball should bounce the other way horizontally. The first portion of this is easy: If the ball has passed the window's right edge, then it should bounce. But it should also bounce if the ball has struck the paddle's right edge, and that is much more complex. As illustrated in Figure 13.6(a), the ball has struck the paddle's right edge if the ball meets all of the following criteria:

- the ball's left edge has crossed (is to the left of) the paddle's right edge,
- the ball's left edge has not gone too far past the paddle's right edge,
- the ball is below the paddle's top, and
- the ball is above the paddle's bottom.

These four criteria appear in the program on lines 30–31.

Figure 13.6: Testing whether a ball has struck a paddle.



Our program simplifies the paddle-ball logic by ignoring an important boundary case, illustrated in Figure 13.6(b): If the ball strikes a corner of the paddle, it will simply continue through the paddle, since the *y*-coordinate of ball's center is outside the range of *y*-coordinates that the paddle occupies. A more industrial-strength Pong program would have more sophisticated logic for bouncing off the paddle's corner. Of course, an industrial-strength Pong would include many more features, too, like keeping score somehow, replacing the ball after it has gone off the window, and restricting the paddle's velocity. Such features are left for you to enjoy adding to the simple Pong program.

Source: http://www.toves.org/books/java/ch13-interact/index.html