USER INTERACTION IN JAVA

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.

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.

```
1  import java.awt.event.*; // for MouseEvent
2  import java.awt.*;       // for Color
3  import acm.graphics.*;   // for GOval
4  import acm.program.*;    // for GraphicsProgram
5
6  public class DropCircles 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         GOval circle = new GOval(mouseX - 10, mouseY - 10, 20, 20);
15         circle.setFillColor(new Color(255, 0, 0));
16         circle.setFilled(true);
17         add(circle);
18     }
19 }
```

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.

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.

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