Object Oriented Programming

Graphical Input and Output

Dr. Seán Russell
sean.russell@ucd.ie

School of Computer Science,
University College Dublin

September XX, 2019

Learning outcomes

After this lecture and the related practical students should...

- be able to draw shapes on the screen
- understand the concept of of callbacks and how these are used with interfaces
- to be able to use callbacks to interact with the user

Table of Contents

Drawing on the Screen

- In the inheritance lecture we saw examples of the use of an abstract class that involved drawing on the screen
- In this lecture, we will focus on this process
- Additionally, we will examine how users can interact with this type of interface using callback methods

Table of Contents

When to Draw

- How will these classes know what code to execute when the screen is drawn?
- We let the classes know by implementing specific methods within our classes, the JVM will then call these methods at the correct times automatically
- This idea is called **inversion of control**
- We create a class that extends a class in the library, in this class we implement the correct methods that tells Java what should be drawn
- Then we only have to tell Java what objects it should draw on the screen

When to Draw

- It is difficult to know when our code should be executed
- The code will need to be executed when the image is first drawn on the screen, whenever the window is moved, if another window is moved in front of it, when the window is resized and in many other situations
- The classes in the library will do all of this for us
- However, this means that we are no longer in control of when our code is executed

Table of Contents

Creating a Window

The first thing we need is somewhere to draw our images, this is done in a window. The steps to create a window and draw on the screen are as follows:

1. Create a window (called a JFrame in Java)
2. Set the size of the window
3. Tell it to shut when the X button is pressed
4. Create a painted component to be displayed
5. Add our painted component to the JFrame
6. Set the JFrame to be visible on the screen

Creating a Window

```
1  public class Window {
2    public static void main(String[] args) {
3      JFrame window = new JFrame();
4      window.setSize(300, 400);
5      window.setDefaultCloseOperation(
       JFrame.EXIT_ON_CLOSE );
6      Picture pic = new Picture();
7      window.add(pic);
8      window.setVisible(true);
9    }
10 }
```

What is Drawn?

- The code shown creates a window and adds a `Picture` object to it
- In order to define the Picture class, we need to know what type of object can be added to the JFrame

add

`public Component add(Component comp)`
Appends the specified component to the end of this container.

Parameters: comp - the component to be added

- The add method shows that a component can be added to a JFrame
- We need to extend the `Component` class or one of its subclasses in the Picture class

JComponent

- The abstract `JComponent` class is the class that most graphical components extend and is a subclass of Component, we will extends this
- Because most of the work of drawing on the screen is taken care of by the JVM, we need to know what method to implement to draw our picture
- The method `paintComponent` is called automatically when the contents of a `JComponent` should be drawn, this is where our drawing code should go

JComponent

paintComponent

`protected void paintComponent(Graphics g)`

Lots of info

Parameters: `g` - the Graphics object to draw on the screen

The Picture Class

- This API excerpt contains a lot of complicated information
- All we really need to know about it is the signature of the method
- `protected void paintComponent(Graphics g)`

```
1 public class Picture extends JComponent {
2   protected void paintComponent(Graphics g) {
3     // Shape drawing code goes here
4   }
5 }
```

- The code that we add to this method will be executed whenever the JVM decides

Table of Contents

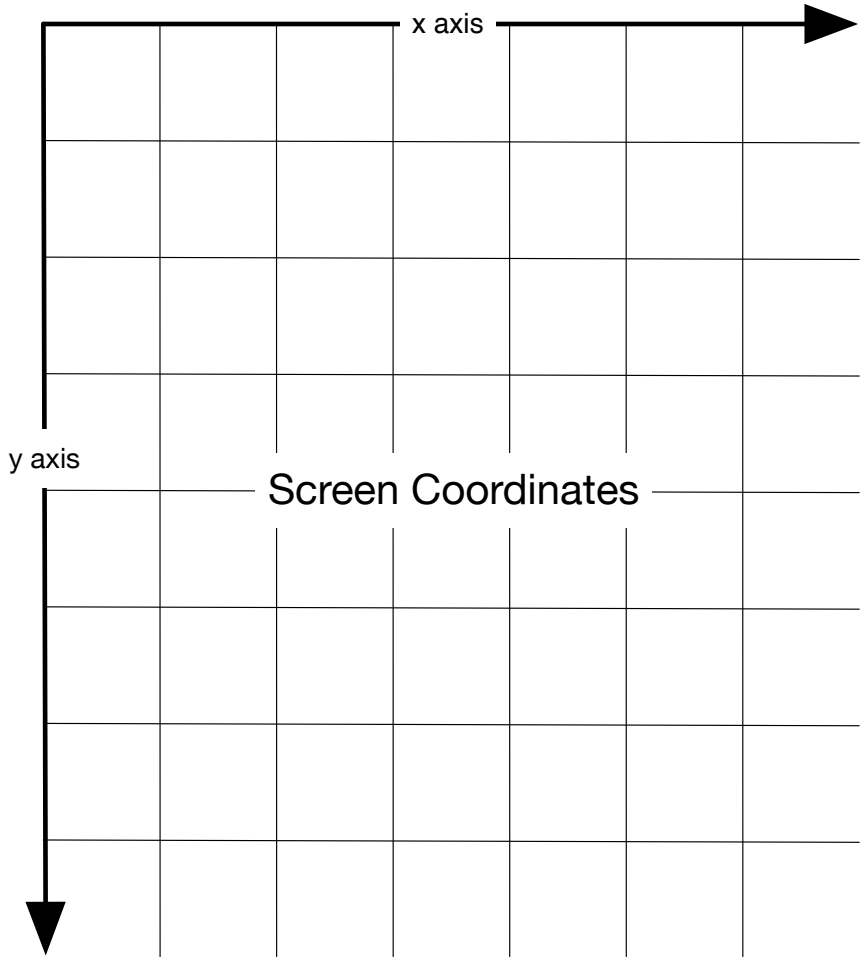
Screen Geometry

- Before we can begin drawing shapes on the screen, we need to understand a little bit more about the how shapes are represented on the screen
- The most important point is to understand the coordinate system that is used in computer graphics
- Older computer screen s worked by aiming electrons at the back of the screen pixel by pixel
- They started one line at a time from left to right
- The first line was at the top of the screen and the last was on the bottom. This entire process would happen many times every second

Screen Coordinates

- Because the process started at the top of the screen and moved down, the coordinate system that was used has the origin at the top left of the screen and the value of y increases as the we move down
- The next slide shows a comparison of this coordinate system with the traditional coordinate system used in normal calculations
- Screen coordinates are counted in pixels, where each pixel is a single dot on the screen
- As we can not draw anything smaller, this means that all of the numbers we use will be integers

Screen Coordinates



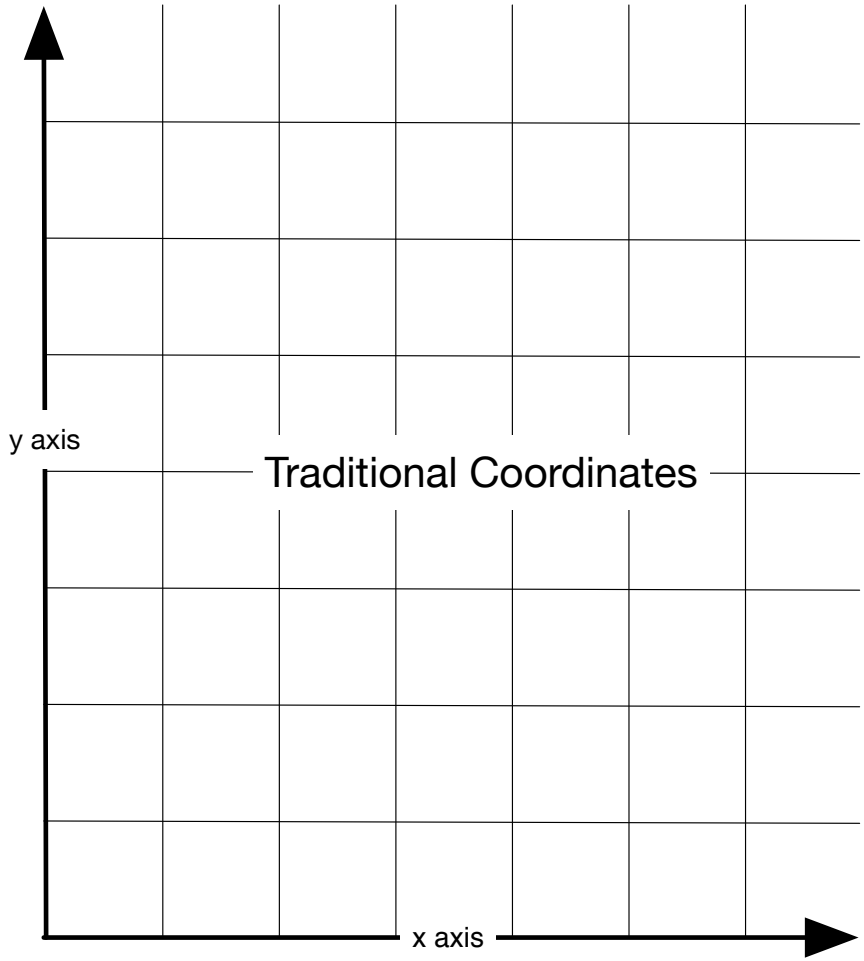


Table of Contents

How To Draw Shapes

- The drawing code is going to be within the method `paintComponent`, now we must look at how we actually draw
- Drawing is done using the parameter that is passed to this method, `Graphics g`
- The graphics class contains many methods that can be used to draw various shapes on the screen, some of these methods are easy to use and others are more complicated

Methods of Graphics

Here is a list of some of the easiest methods to use when drawing on the screen.

- void drawLine(int x1, int y1, int x2, int y2)
- void drawOval(int x, int y, int width, int height)
- void drawPolygon(Polygon p)
- void drawRect(int x, int y, int width, int height)
- void fillOval (int x, int y, int width, int height)
- void fillRect (int x, int y, int width, int height)
- void fillPolygon (Polygon p)

Changing Colour

- Additionally, the colour that these shapes are drawn in can be changed by calling the method setColor, which takes a color object as a parameter
- Typically, we will use colors that are already represented as constants in the `Color` class, such as Color.black, Color.green, Color.blue
- However, we can also create our own colours by creating a new color object by passing in values for red, green and blue to the constructor

Design Methodology

- We are going to approach this topic in an OOP way, our picture will be made up of a number of smaller drawings and each of these represented by a class
- These classes will extend `Abstract2DDrawing`
- Inside the draw method of each class we will add the code to draw each smaller drawing
- We will start with a very basic drawing of a building shape on a skyline, this can be represented by a simple rectangle
- Our basic building will be 200 pixels high and 50 pixels wide
- The abstract 2D drawing class contains a constructor to set the x, y location of the drawing

Relative Coordinates

- If we want to place a building at position (50, 50), then the value of x and y are set to 50
- When we draw the rectangle, we use the values of x and y to call the method
- The only code we need in our draw method is `g.fillRect(x, y, 50, 200);`.
- Drawing multiple shapes in a drawing, we will always need to use values that are relative to `x` and `y`
- For example, the position of a shape might be x + 20, y + 10
- This means that whatever value we set the variables x and y, the parts of the image will be moved

The Building Class

```
1 public class Building extends
     Abstract2DDrawing {
2   public Building(int x, int y) {
3     super(x, y);
4   }
5   public void draw(Graphics g) {
6     g.fillRect(x, y, 50, 200);
7   }
8 }
```

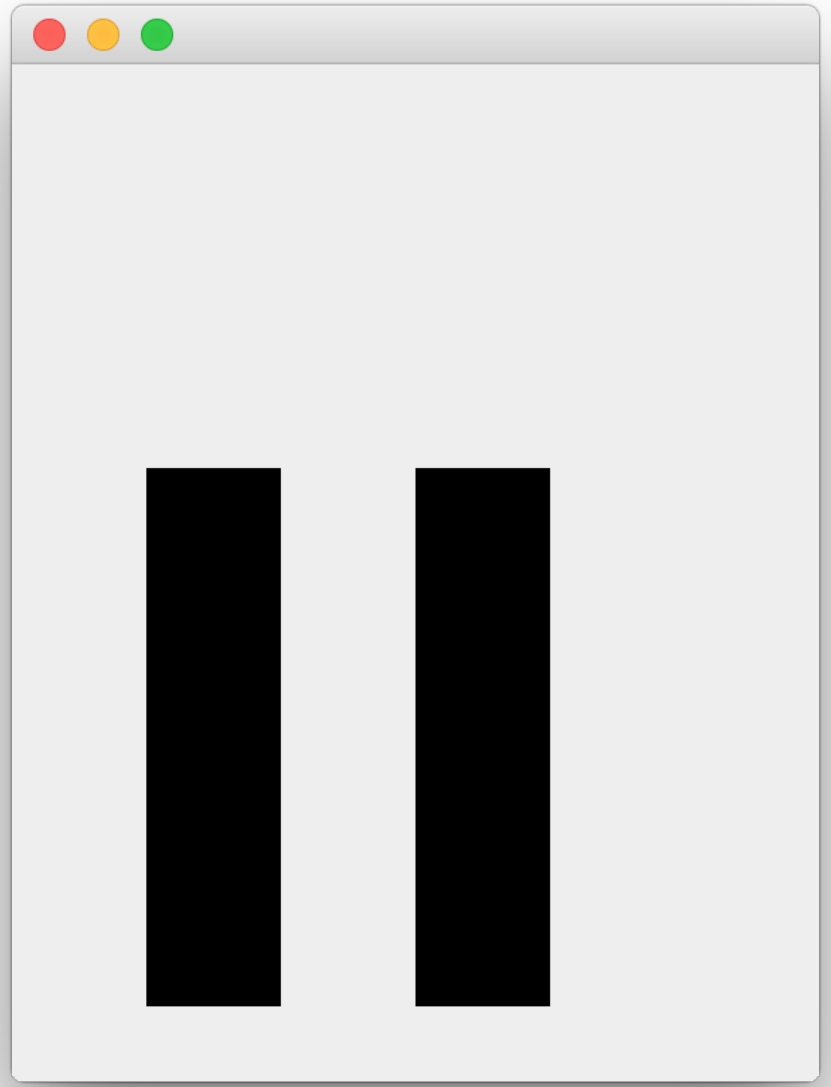
Drawing the Building

- Now that we have completed this class, we can finally add the required code to the paintComponent method in the picture class to add our buildings
- The nice thing about the strategy that we have used is that based on the same class, we can add the same drawing multiple times by creating multiple objects

Drawing Building Objects

```
1 protected void paintComponent(Graphics g) {
2   Building b = new Building(50, 150);
3   b.draw(g);
4   Building b2 = new Building(150, 150);
5   b2.draw(g);
6 }
```

The Drawing



Complicated

- Using a class to represent a building is more complicated than simply drawing the shapes
- However, as the individual pieces get more complicated, the use of this simplifies the process
- We will show this by making our picture more complicated, adding windows to the image will make these shapes look more like buildings
- Windows can be added by drawing smaller rectangles in a different colour in a rows on top of the rectangle we have already drawn
- As our building is 50 pixels wide we can easily fit 3 10 by 10 windows on each floor and as it is 200 pixels high we can easily add 12 floors of windows

Drawing Windows

- Drawing a single floor of windows can be done using a loop that executes 3 times, we just need to use the index of the loop to calculate the position of the window
- There should be a gap of 5 between each window, so we can calculate the x coordinate of the window by multiplying i * 15 (width of window + gap) and adding 5 (gap at beginning)
- However, because all of these buildings can be moved, the coordinates of the window must be relative to the position of the building itself, we must also add x

Drawing Windows

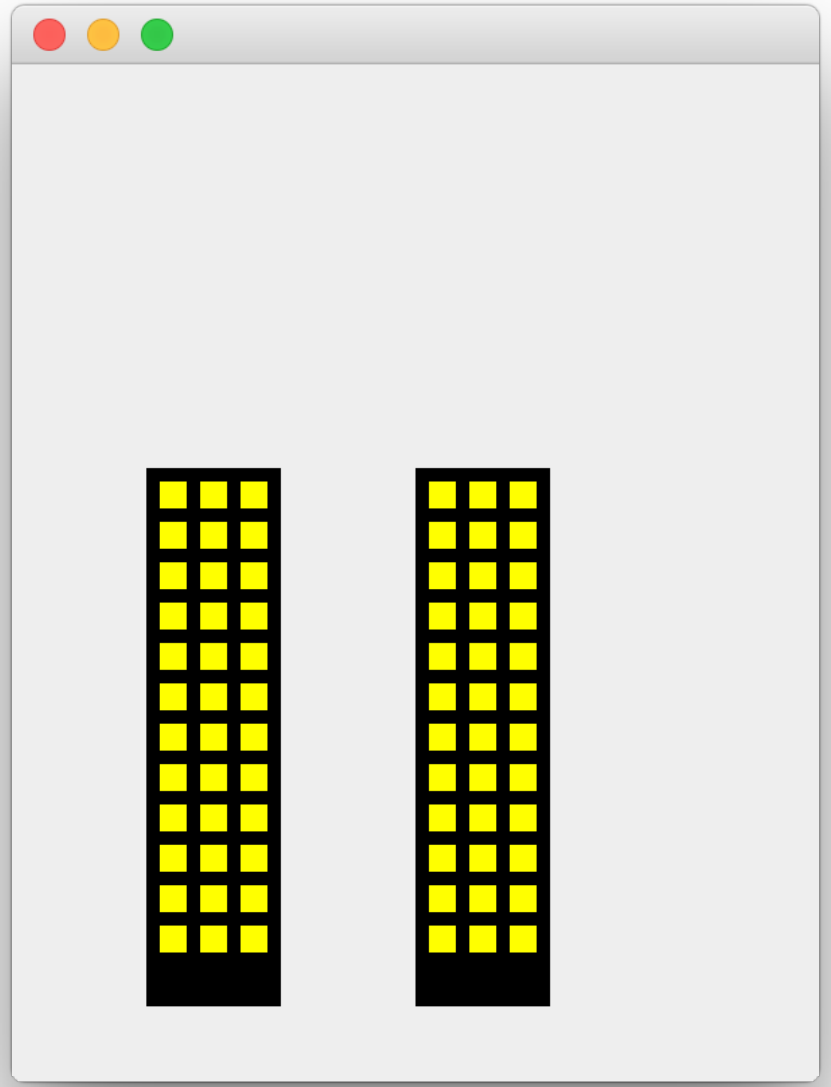
- The same process is used to calculate the y coordinate of each window, so we end up with nested loops containing something like this:
  `g.fillRect(5 + x + j * 15, 5 + y + i * 15, 10, 10);`

- But before we draw the window, we must first change the colour. Additionally, we must also make sure the colour is correct before we draw the main rectangle

Updated draw method in Building class

```
1  public void draw(Graphics g) {
2    g.setColor(Color.black);
3    g.fillRect(x, y, 50, 200);
4    g.setColor(Color.YELLOW);
5    for (int i = 0; i < 12; i++) {
6      for (int j = 0; j < 3; j++) {
7        g.fillRect(5 + x + j * 15, 5 + y + i *
       15, 10, 10);
8      }
9    }
10 }
```

The Updated Drawing



Benefits of Drawing with Classes

- We can also extend the class to make a slightly different picture
- To do this we create a new class that extends the Building class called `RoofBuilding`
- This will be a slightly bigger building because we are adding a roof to the top, so in the constructor we move the building down by adding 20 (the height of the roof) to the value of y when we pass it to the super constructor

Benefits of Drawing with Classes

- To draw the roof we have to override the draw method of the building class
- However, because we have already written the code to draw the building, we do not want to write it again
- Instead we first call the draw method of the superclass
- Then we can add the code to draw our roof

The RoofBuilding Class

```
1  public class RoofBuilding extends Building{
2    public RoofBuilding(int x, int y) {
3      super(x, y+20);
4    }
5    public void draw(Graphics g) {
6      super.draw(g);
7      g.setColor(Color.black);
8      Polygon p = new Polygon();
9      p.addPoint(x, y);
10     p.addPoint(x, y-20);
11     p.addPoint(x+50, y);
12     g.fillPolygon(p);
13   }
14 }
```

The Updated Drawing

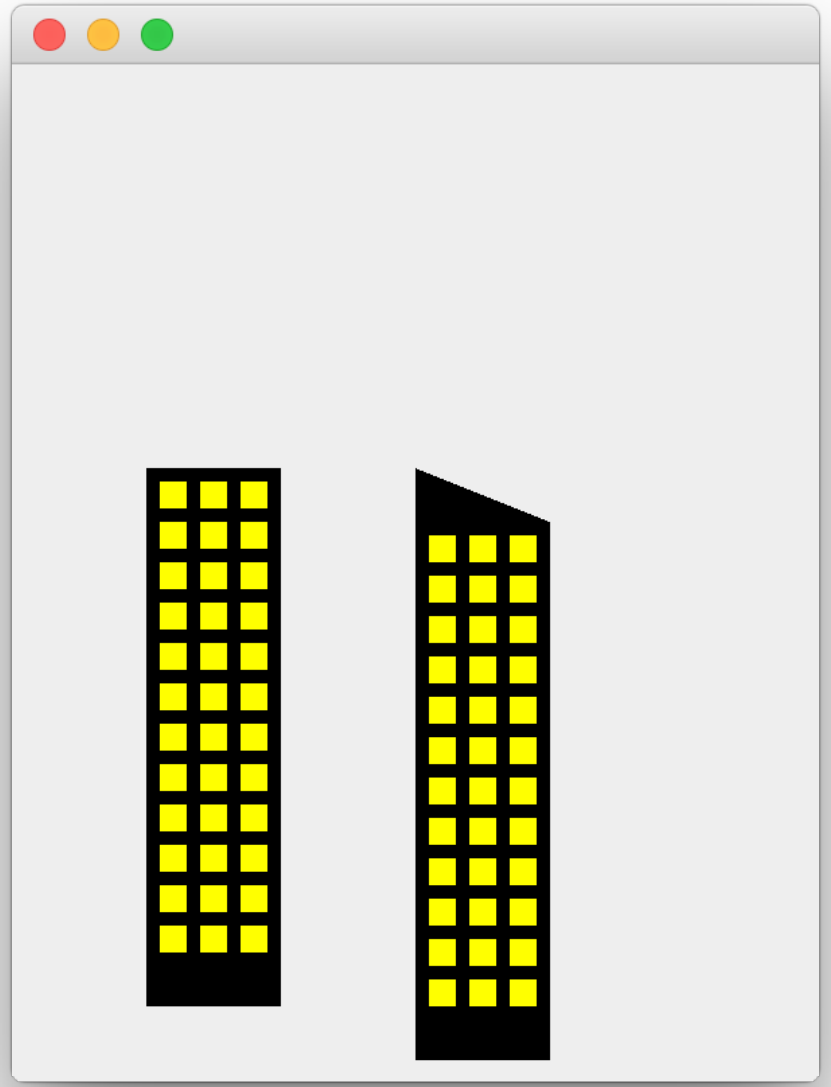


Table of Contents

When to Execute Code?

- It is difficult to know when the mouse has been used
- Parts of the JVM that we can use to be told whenever different things happen to with the mouse
- This works based on the concept of **callback** methods
- A callback method is a method that we implement knowing that it will be called whenever a particular event happens, such as the mouse being clicked
- The functionality of callbacks is implemented using interfaces in Java
- Classes that implement these interfaces are often called **listeners**

MouseListener

We need to implement the `MouseListener` interface from the package `java.awt.event` The mouse listener interface requires that the following methods are implemented:

- `void mouseClicked(MouseEvent e)`
  This method is called whenever the mouse is pressed down and released
- `void mousePressed(MouseEvent e)`
  This method is called whenever the mouse is pressed down
- `void mouseReleased(MouseEvent e)`
  This method is called whenever the mouse is released

MouseListener

We need to implement the `MouseListener` interface from the package `java.awt.event` The mouse listener interface requires that the following methods are implemented:

- `void mouseEntered(MouseEvent e)`
  This method is called whenever the mouse moves inside the window

- `void mouseExited(MouseEvent e)`
  This method is called whenever the mouse moves outside the window

Implementing the MouseListener Interface

- Mostly, we will only add code to one or two of the methods and the rest will be left empty
- We will implement some code that will be executed whenever the mouse is clicked, so we will add our code to the first method
- What we will make this code do is turn the lights on and off in our building drawing, however this will require us to add and change some code in the Picture and Building classes first

Lights in Buildings

- First we have to change the code of the Building class so that the lights can be turned on and off, this can be done by changing the colour that the windows are drawn in
- So we need to add an instance variable that holds the value as to if the lights are on or off
- Because this is variable has only two possible values, it should be implemented as a boolean variable
- So we add `private boolean lights = true;` to the Building class (and its subclasses automatically)

Lights in Buildings

- Secondly, we need to choose the colour of the windows based on the value of the variable `light`
- The most sensible and understandable is to have the lights be turned on whenever the value of lights is `true`
- So we add an if-else statement that sets the colour before we draw the windows to either yellow for lights on or blue for lights off.

Lights on or off

```
1 if(lights){
2   g.setColor(Color.YELLOW);
3 } else {
4   g.setColor(Color.BLUE);
5 }
```

Lights in Buildings

- Lastly, we need to add some way for this to be changed from outside of the class
- So we add a method to the class that passes in a boolean value and changes the value of lights to match

Set value of boolean

```
1 public void setLights(boolean b){
2   lights = b;
3 }
```

Updating Picture class

- Next we need to have some way in the Picture class for turning the lights on and off in the buildings
- However, in the code we designed earlier the Building objects are created whenever the paint component method is called
- We need to instead make these objects instance variables of the class if we want to be able to change them from other methods
- We also need to add a method to allow the lights to be turned on and off
- The method is basically the same signature but in the code we just call the method on each of the building objects

Updated Picture Class

```
1  public class Picture extends JComponent {
2    Building b = new Building(50, 150);
3    RoofBuilding b2 = new RoofBuilding(150,
       150);
4
5    protected void paintComponent(Graphics g) {
6      b.draw(g);
7      b2.draw(g);
8    }
9
10   public void setLights(boolean l){
11     b.setLights(l);
12     b2.setLights(l);
13   }
14 }
```

Ready for mouse Listener

- Now we are finally ready to implement our mouse interaction, we will do this in a class called `LightListener`
- For the light listener to be able to call the method of our picture object, it must have a reference to it
- So we add a constructor that requires a picture object and an instance variable of a picture object
- We also need to know if the lights were on or off already, so we need an instance variable to remember what the value of lights was, we will call this `lightOn` and set the initial value to true because the lights will be on when the picture is first drawn

Implementing for mouse Listener

- In the mouseClicked method, we add code to change the value of the variable `lightOn`
- Second we add the code to call the set light method in the picture object to the new value of `lightOn`
- However, the JVM does not know that the picture has changed and needs to be drawn again
- Lastly we call a method on the picture object called `repaint`, this tells the system that the picture object needs to be drawn again and the system will call the required methods

LightListener Class

```
1  public class LightListener implements MouseListener {
2    private Picture picture;
3    private boolean lightOn = true;
4    public LightListener(Picture p){
5      picture = p;
6    }
7    public void mouseClicked(MouseEvent e) {
8      lightOn = !lightOn;
9      picture.setLights(lightOn);
10     picture.repaint();
11   }
12   public void mousePressed(MouseEvent e) {}
13   public void mouseReleased(MouseEvent e) {}
14   public void mouseEntered(MouseEvent e) {}
15   public void mouseExited(MouseEvent e) {}
16 }
```

What Next?

- So far we have changed the code to allow the lights to be turned on and off and we have defined a class that knows what to do when the mouse is clicked
- This leaves two important tasks still to be completed
- First we need to actually create an object based on the `LightListener` class
- Secondly we need to tell the window that we want to be told whenever one of these events actually happens
- This must be done in the `Window` class

Updated Window Class

```
1  public class MyWindow {
2    public static void main(String[] args) {
3      JFrame window = new JFrame();
4      window.setSize(300, 400);
5      window.setDefaultCloseOperation(
       JFrame.EXIT_ON_CLOSE );
6      Picture pic = new Picture();
7      window.add(pic);
8      LightListener ll = new
       LightListener(pic);
9      window.addMouseListener(ll);
10     window.setVisible(true);
11   }
12 }
```

Important Parts

- On line 8 of the above example, we create an object (`ll`) based on the light listener class
- More importantly on line 9 we tell the `JFrame` that we are interested in hearing about mouse events by adding the light listener object as a mouse listener
- This now means that whenever the system detects that a mouse click has happened within our window, the `mouseClicked` method in the light listener object will be called and the lights will be changed

The Lights Turned Off

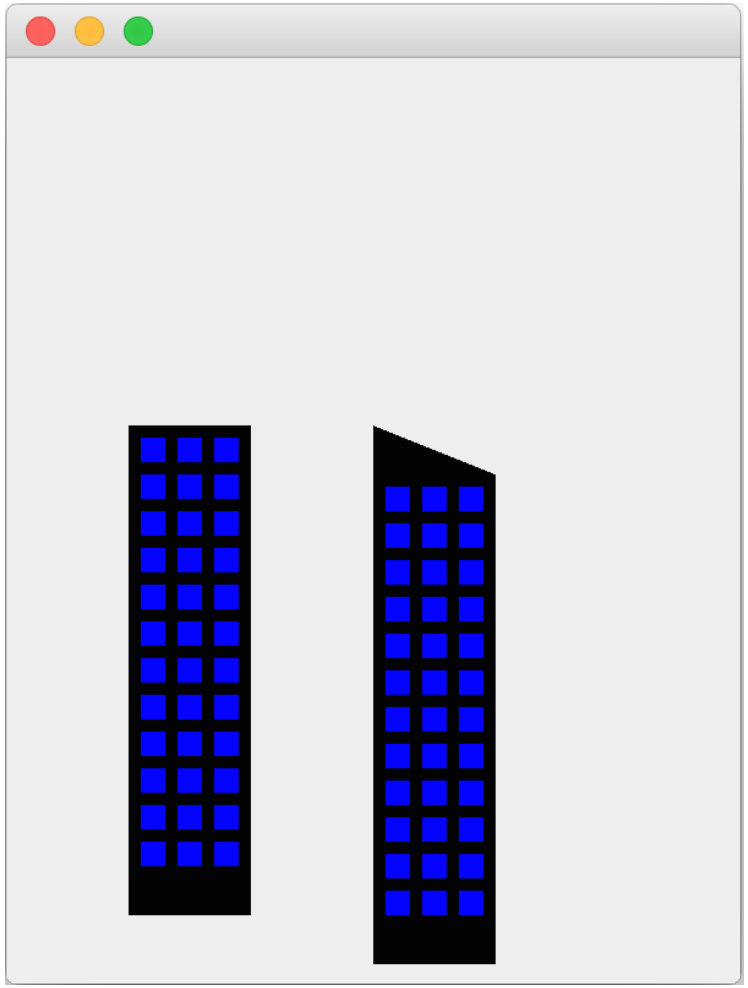


Table of Contents

Mouse Motion Listener

- It is also possible to use the motion of the mouse to control our applications
- This is done using by implementing the `MouseMotionListener` interface
- This interface has only two method, `void mouseDragged(MouseEvent e)` and `void mouseMoved(MouseEvent e)`
- Once created a mouse motion listener class can be added as a listener using the method `void addMouseMotionListener(MouseMotionListener m)`

Mouse Motion and Click Listener

- If we are required to implement listeners for both the mouse clicks as well as motion of the mouse, we can simplify the process by using the `MouseInputAdapter` class
- This is an abstract class that implements both mouse listener interfaces and provides an empty implementation for each method
- We simply need to extend the class and override the methods that we want to use
- The object created based on this class needs to be added both as a mouse listener and as a mouse motion listener

Table of Contents

Keyboard Callbacks

- We can use the same technique to interact with the keyboard
- This allows us to read key presses rather than reading the text afterwards
- This is done with the KeyListener interface
- The interface defines the following methods
  - `void keyTyped(KeyEvent e)`
    This method is called whenever a character key (not function keys) is pressed and released
  - `void keyPressed(KeyEvent e)`
    This method is called whenever any key is pressed down
  - `void keyReleased(KeyEvent e)`
    This method is called whenever any key is released

KeyListener Events

- Just as in the mouse listener interface, a single parameter is passed to the method when it is called
- In the mouse example, we ignored this parameter as we did not need to know any details about the individual clicks
- However, when using the keyboard to interact, we will want to know which of the keys has been pressed and this information is contained in the `KeyEvent` parameter

Table of Contents

Key Events

- Every key event object contains a number that represents the key that was pressed
- We can access this number by calling the method `getKeyCode`
- This method requires no parameters and returns an int value

KeyEvent - getKeyCode

```
public int getKeyCode()
```

Returns the integer keyCode associated with the key in this event.

Returns: the integer code for an actual key on the keyboard.

Comparing Key Events

- For each key on the keyboard, there is a constant in the KeyEvent class that contains the value
- These constants have named like `VK_LEFT` or `VK_A` and can be looked up in the documentation of the KeyEvent class
- This allows us to write code that does a different action depending on what key was pressed
- For example, if we wanted to move an object when the left key was pressed we would add an if statement with the condition `e.getKeyCode() == KeyEvent.VK_LEFT`.

Drawing a Moon

- We create a new class called Moon that extends the Abstract2DDrawing class
- This class will just add a circle wherever it is positioned

Moon

```
1 public class Moon extends Abstract2DDrawing {
2   public Moon(int x, int y) {
3     super(x, y);
4   }
5   public void draw(Graphics g) {
6     g.setColor(Color.GRAY);
7     g.fillOval(x, y, 50, 50);
8   }
9 }
```

Moving the Moon

- We need to add a method that changes the position of the moon in the x and y directions
- This method requires two parameters, the first to tell us how much to move in the x direction and the second to tell us how much to move in the second direction
- The values in these parameters are then passed to the move method in Moon object

Added to Picture class

```
1 public void moveMoon(int x, int y){
2   moon.move(x, y);
3 }
```

Creating Moon Listener

- Now we have added an object that can be moved in the picture and a method that we can call to move it
- Now we need to create our key listener class
- Just as with the light listener class, we will require an instance variable to store a reference to the picture object and constructor to pass the object
- We will implement the `keyPressed` method and have the moon move any time one of the direction keys are pressed

Which way to Move?

- To know which way to move the moon, we have an if statement for each of the values `VK_LEFT`, `VK_RIGHT`, `VK_UP` and `VK_DOWN`
- Within the body of each of these if statements, we will call the `moveMoon` method and pass in the parameters required to have the moon move 10 pixels in the correct direction
- Finally, we also need to tell the picture that it needs to be repainted, this could be done individually after the `moveMoon` method was called, but it is easier to just add it once after the if statements

Completed Key Listener

```
1  public class MoonListener implements KeyListener {
2    private Picture picture;
3    public MoonListener(Picture p){
4      picture = p;
5    }
6    public void keyTyped(KeyEvent e) {}
7    public void keyReleased(KeyEvent e) {}
8    public void keyPressed(KeyEvent e) {
9      if(e.getKeyCode() == KeyEvent.VK_LEFT){
10       picture.moveMoon(-10, 0);
11     } else if(e.getKeyCode() == KeyEvent.VK_RIGHT){
12       picture.moveMoon(10, 0);
13     } else if(e.getKeyCode() == KeyEvent.VK_UP){
14       picture.moveMoon(0, -10);
15     } else if(e.getKeyCode() == KeyEvent.VK_DOWN){
16       picture.moveMoon(0, 10);
17     }
18     picture.repaint();
```

Last Step

- Again, we need to complete the final two steps or our code will never be executed
- We still must create an object based on the MoonListener class and tell the system that we are interested in hearing about key events and that the methods of the MoonListener object should be called

Adding Moon Listener

```
1 MoonListener ml = new MoonListener(pic);
2 window.addKeyListener(ml);
```