Part IV

Event based programming – Events

Chapter 42

Static GUI – Frame, components

Up till now, all interaction with the program was through console (or file) I/O. Now this is extended with graphics, ranging from displaying things on the screen to interaction using the mouse or (arrow) keys on the screen: the Graphical User Interface (GUI).

This chapter concerns simple displaying of predefined components on the screen – in further chapters interaction using these components, e.g., a clickable button, will be addressed.

42.1 Aim

We want to display graphical things on the screen.

42.2 Means

We use classes from the Java API to make screen things.

There are two sets of GUI classes in Java: the basic Abstract Window Toolkit, in packages starting with `java.awt`, and the more versatile Swing classes, in packages starting with `javax.swing`. The Swing classes have more or less replaced the AWT classes and we will treat Swing in this book. Nevertheless, AWT classes are still available to support older programs. The names Swing classes start with a `J`, to distinguish them from their AWT counterparts. Some AWT classes, such as *Color* and the event classes are not replaced in Swing.

There are three kinds of GUI classes.

Component classes The things you see on the screen are all built from classes that are subclasses of the class *Component*. Therefore, we will call them components. For example *JButton*, a clickable button, and *JTextField*, a box where text can be displayed or entered by the user.

Container classes These classes, which are also components, can contain other GUI components. For example *JFrame*, a window, and *JPanel*, a rectangualr section of a window that may contain other components.

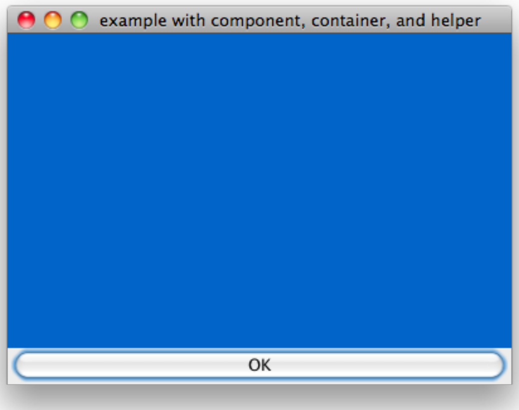
Helper classes These are not components and are not directly visible on the screen. They provide additional functionality for the GUI. For example *Color* and *LayoutManager*.

42.2.1 Using components

GUIs are built starting by creating a frame from the class *JFrame*, the outermost container. This provides a window that appears on the screen if it is declared *visible* by calling `setVisible(true)`. By default, a JFrame is not visible. All other components are by default visible, so it is not necessary to call setVisible on them.

The following program creates a window and puts a blue panel and a button with the text "OK" in it. The result of executing this program is shown below the program.

```
import java.awt.∗;
import javax.swing.∗;

class FrameDemo {

    void makeFrame() {
        JFrame frame = new JFrame (”example with component, container, and helper ”);
                                                  // creates window: JFrame

        JButton button = new JButton(”OK”); // creates component: JButton
        frame.add(button, BorderLayout.SOUTH);
                                       // puts component in container: button in frame

        JPanel panel = new JPanel(); // creates another component
        frame.getContentPane().add(panel); // puts panel in frame

        Color color = new Color(0, 100, 200); // makes helper object: color blue
        panel.setBackground(color); // colors background panel

        frame.setSize(400, 300);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                                                  // closing window exits program
        frame.setVisible(true);

    }

    public static void main(String[] args) {
        new FrameDemo().makeFrame();
    }
}
```



Remarks

- Here, we just consider creating a button: programming a reaction to the clicking of a button will be treated in chapter **??**.
- In older versions of Java, you couldn't directly add a component to a JFrame. An intermediate object, the *content pane*, had to be provided. Therefore, in older programs you will see `frame.getContentPane().add(panel)` etc. This still works, but the code in the example is simpler and clearer.
- The class *Color* is a helper class. Objects of this class represent colors that can be attributed to components. See section 42.3.2 for more information.

42.2.2 Positions and sizes

The position on screen and the size of components are measured in *pixels*. Many methods have two parameters to determine position or size. The first parameters always refers to the *horizontal* size or position, the second parameter refers to the *vertical size or position*. E.g., `frame.setSize(120, 100)`, sets the size of the window to 120 wide and 100 pixels tall.

42.2.3 Layout managers

The space inside a frame is organized using containers in which components are placed. Note that containers are also components, which means that containers can be placed inside other containers. In Java, the placing of the components in the window is performed during run-time. The reason for this is that the optimal size of a component depends on the current computer (size of text in buttons, e.g.) and furthermore it allows the user to resize windows on the fly.

The strategy on how to size and where to place components is determined by an object called *layout manager*. There are several types of layout managers, each defined in a different class, and all subclasses of *LayoutManager*. Examples:

- `FlowLayout` (the default layout manager of a `JPanel`) puts components in "reading order": it fills a horizontal row with components; when the row is filled, it continues with a new row below the first.
- `BorderLayout` (the default layout manager of a `JFrame`) puts component in one of the four sides of the container, or in the centre.
- `null`: no layout manager (see below).

Every component has a default layout manager. When you want to change the layout manager of a component, you create a new object of the desired class and tell the component to use that object as its layout manager. Example (suppose `comp` is the component):

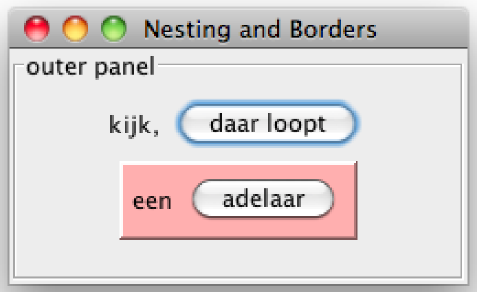
```
FlowLayout layoutmanager = new FlowLayout(); // this is the object
comp.setLayout(layoutmanager);  // from now on, FlowLayout will be used for comp
```

or, in one line,

```
comp.setLayout( new FlowLayout() );
```

In the example below, we have a JFrame with two JPanels. one inside the other. To better see where the JPanels are, we put a border around them. *Border* is a helper class, it is not a subclass of component. For reasons of efficiency, Border object are created by means of a so-called factory class, not via `new`.

```
// shows nested panels with borders
import java.awt.*;
import javax.swing.*;
import javax.swing.border.Border;

class NestingAndBorders {
    JFrame frame;
    JPanel outer = new JPanel();
    JPanel inner = new JPanel();
    JLabel a = new JLabel("kijk,");
    JButton b = new JButton("daar loopt");
    JLabel c = new JLabel("een");
    JButton d = new JButton("adelaar");

    NestingAndBorders() {
        Border schuinrandje = BorderFactory.createRaisedBevelBorder();
        Border lijntjemetnaam = BorderFactory.createTitledBorder("outer panel");

        frame = new JFrame("Nesting and Borders");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(250, 150);

        outer = new JPanel();
        outer.setBorder(lijntjemetnaam);
        frame.add(outer);

        outer.add(a);
        outer.add(b);
        outer.add(inner);

        inner.add(c);
        inner.add(d);
        inner.setBorder(schuinrandje);
        inner.setBackground(Color.PINK);

        frame.setVisible(true);
    }
```



No layout manager

It is often difficult to get the position of your components exactly the way you want when you use layout managers. Partly, this is a good thing, because the whole idea of layout managers is that the positioning is automated. Nevertheless, like every automation, it can sometimes be annoying. Therefore, we give an example of how to avoid the layout manager altogether.

The method *setLocation(int x, int y)* changes the position of a component. Note that positions are specified relative to the top left corner of the container. x is measured from left to right, y is measured from top to bottom. With *setSize* you can change its size. See also the Java API.

```java
import java.awt.*;
import javax.swing.*;
import java.util.*;

class NullLayoutExample {
    Scanner sc = new Scanner(System.in);

    void makeExample() {

      JFrame frame = new JFrame ("The frame"); // makes frame
      JPanel panel = new JPanel(); // makes panel
      frame.add(panel); // puts panel in frame

      panel.setLayout(null); // disables layout managing for panel

      JButton button = new JButton("OK"); // creates button
      button.setSize(button.getPreferredSize()); // of standard size

      panel.add(button);
      button.setLocation(20, 120); // puts button 20 pixels from the left
                                   // and 120 pixels from the top of the panel

      frame.setSize(400, 300);
      frame.setVisible(true);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // ask repeatedly for horizontal position and place button there
      int xpos;
      while (true) {
          System.out.println("Where do you want the button?");
          xpos = sc.nextInt();
          button.setLocation(xpos, 120);
       }

    }

   public static void main(String[] args) {
        new NullLayoutExample().makeExample();
   }
}
```

42.3 Some important GUI classes

42.3.1 Some Swing Components

class	description	some methods and constructors

JComponent	parent class of all Swing components; you don't make instances of this class; it contains many methods that are shared by all components	paintComponent(Graphics g)
		int getWidth()
		int getHeight()
		setSize(int w, int h)
		repaint()
		setBackground(Color c)

JFrame	window with title bar etc.	add *(inherited from Container)* setDefaultCloseOperation setVisible(boolean b) *(inherited from Component)*
JPanel	empty component, used for drawing or as a placeholder of other components	add *(inherited from Container)*
JButton	clickable button	JButton(String name)
JTextField	editable line of text	setText(String text) String getText()

JLabel	a line of text that is not editable and/or an image	JLabel(String text)
		JLabel(Icon image)

JApplet	component that can be added to an internet page; inherits from JPanel	init() start(0

Box	a JPanel-like container with a BoxLayout; this places components in one horizontal or vertical row	Box(BoxLayout.X_AXIS)
		Box(BoxLayout.Y_AXIS)

42.3.2 Some helper classes

class	description	some methods / fields / constructors
Color	represents a color; uses the RGB color model by default; contains a lot of constants for predefined colors	Color.RED, Color.GREEN, etc. Color(int red, int green, int blue) brighter(), darker()
FlowLayout	layout manager that places the components in "reading order"	
GridLayout	layout manager that places components in a rectangular grid components are added from left to right and top to bottom	GridLayout(int rows, cols)

BoxLayout	layout manager that places the components in horizontal or vertical row, as a line of words or as a page of words	BoxLayout(container, BoxLayout.X_AXIS)
		BoxLayout(container, BoxLayout.Y_AXIS)

BorderFactory	creates borders	BorderFactory.createTitledBorder(String title)
		BorderFactory.createLineBorder(Color color)
		BorderFactory.createRaisedBevelBorder()
		and many more

Border	often only implicitly used (see second call)	*someComponent*.setBorder(Border border) *someComponent*.setBorder(BorderFactory.create...)

Chapter 43

User triggered activity – device, source, low level events, listeners

Up till now, the initiative for execution has been with the program. The program could prompt the user to interact with it by requiring console input and waiting. Then the user, from that moment on, could give an input and, after pressing the `Enter` key, the program would proceed, possibly prompting the user again later. Now a different form of user interaction is introduced, where the program does not prompt the user but is responsive to the user, i.e., at any moment the user, from outside the program, can trigger the execution of a chosen activity. All triggers are captured and eventually reacted on, also if a trigger occurs while the reaction to an earlier one is still in progress, but the reactions corresponding to the triggers are executed sequentially, i.e., there is no concurrency.

43.1 Aim

We want to enable the user to decide on the moment to trigger the execution of a chosen activity. In particular, the user may trigger a new reaction while the current one is still being executed. The current execution should not be interrupted, but after it terminates, the new reaction should be executed as soon as possible.

43.2 Means

We use the *event mechanism* in combination with the Swing GUI (Graphical User Interface). Actions by the user, with a *input device*, the mouse or the keyboard, produce *events*. The event mechanism processes the events, eventually leading to the start of a corresponding reaction that we program. The event mechanism provides scheduling of the reactions. The reactions are executed sequentially, i.e., there is just one active program counter.

In this chapter mouse and key events are considered.

The interaction with the user is through the *Graphical User Interface* (GUI). Several Java *libraries*, collections of classes or interfaces, are available for GUI programming by writing `import <name of the library>;` at the start of the program.

The mouse makes for the most direct explanation, so it is used to exemplify the approach. We write a program that responds to pressing the mouse button on a panel on the screen by putting `Mouse pressed` on the console output as the response to a mouse press on a certain area on the screen.

43.2.1 Programming the means for the user to perform actions that generate events: the source

GUI events are always originating from a screen object, the *event source*. There are various classes that can act as a source, provided as subclasses of class `JComponent` in the `javax.swing.*` library. A source is often called a component.

A device is used to trigger the source to produce an event.

In the example, we use an object of the subclass `JPanel` of `JComponent` as source, providing the screen area to perform the mouse actions on.

User actions for the mouse are moving the mouse and pressing and releasing the mouse button. As the result of user actions with the mouse, the `JPanel` produces several kinds of *event*s. An event is itself an object (hence sometimes called event object), here of class `MouseEvent`. Apart from having a type, like `MouseEvent`, in the event object information is stored that can be used in the further processing of the event. For example, what is the source object and wat is the specific device action, e.g., pressed or released, is stored in the object. We consider the following group of `MouseEvent`s, distinguished by the device actions (later more groups of events will be introduced): `MouseEvent`s with device action information mouseClicked, mouseEntered, mouseExited, mousePressed and mouseReleased.

We only use the mousePressed kind of `mouseEvent` in the example, but as will be seen below, we have to be aware of the other cases as well.

A more detailed explanation of the processing by the event mechanism of events is given in the execution model, Section 5.3.

So the first ingredient for our program is the source: class `JPanel`.

43.2.2 Programming the reaction to events: the listener

The reaction to an event is *handled* (executed) by an object: an *event listener*. We program the reactions to events as methods, so-called *event handler* methods. The Java event mechanism provides the connection between an event and activating the corresponding handler: hence the handler names must be fixed. For a group of events, a listener interface provides the handler name for each type of event. The code for the listener interfaces is provided in the library `javax.awt.event.*`.

The listener interface for our group of `MouseEvent`s is `MouseListener` (later more listeners, for other groups of events, will be introduced).

In the API we see that `MouseListener` provides the handlers for our group of events (this is why above the events with this device action information in the group where presented).

```
public void mouseClicked(MouseEvent e);
public void mousePressed(MouseEvent e);
public void mouseReleased(MouseEvent e);
public void mouseEntered(MouseEvent e);
public void mouseExited(MouseEvent e);
```

We program the desired reactions to the `MouseEvent`s in a class of our own naming, `MousePressedReporter`, that implements `MouseListener`.

We will only use events with mousePressed as device action information and thus only implement the `mousePressed` handler with real behavior. Because all methods of an interface must be implemented, we just provide the required implementation for the other handlers as dummies: {}.

So the second ingredient for our program is the listener: class `MousePressedReporter`.

The numbers in the comments (more in the program parts to come) provide a kind of checklist to check whether all steps to provide source and listener, and their connection, are present.

```
//reacts to mousePressed event by putting ''Mouse was pressed'' on console
import javax.swing.*;
import java.awt.event.*;

class MousePressedReporter implements MouseListener {//2a definition of listener as MouseListener

    public void mouseClicked(MouseEvent e) {};;//all unused handlers
    public void mouseEntered(MouseEvent e) {};//have to be implemented too:
    public void mouseExited(MouseEvent e) {};//here just with dummy bodies {}
    public void mousePressed(MouseEvent e) {
            System.out.println("Mouse was pressed");//2b implementation of reaction by listener
    }
    public void mouseReleased(MouseEvent e) {};//one more unused handler
}
```

43.2.3 Programming the instantiation and combination of source and listener

The events originating from the source should be be handled by the handler methods from the listener. We therefore program that the listener is added to the source, also called *registering* the source with the listener.

To instantiate and combine source and listener, and perform some further preliminaries, we write a class `MouseEventsDemo` with the following content.

We instantiate `JPanel` as source `panel`.

We instantiate `MousePressedReporter` as listener `mousePressedReporter`.

We register the source with the listener using the method `addMouseListener` that class `JPanel` provides.

```
panel.addMouseListener(mousePressedReporter);
```

The effect is that the events generated by that source are handled by the handler methods of that listener.

We provide a window on the screen to put the panel in by instantiating the class `JFrame`. The code for the class `JFrame` is provided in the library `javax.swing.*`. We set some properties of frame, notably providing the exit-on-closure. To end a GUI program, the `x` on the window can be clicked. This click generates an event that causes the program to exit, provided the property of the frame is set to this effect. Hence the line `frame.setdefaultCloseOperation(JFrame.EXIT_ON_CLOSE);`. This is another manifestation of the event mechanism.

We add the panel to the frame using the method `add` that class `JFRame` provides. This results in putting the panel in the window on the screen - in fact, only the panel is visible, as it completely fills the window.

Finally, `frame.setVisible(true);` makes the window with the panel in it appear on the screen, enabling the event driven execution.

The code for the `MouseEventDemo` is then as follows; it uses the source class `JPanel` from the `javax.swing.*` library and the listener class `MousePressedReporter` that we wrote through extending the class `MouseListener` from the `javax.awt.event.*` library, coding the reaction by overriding the handler MousePressed.

```
//puts together source panel and listener mousePressedReporter

import javax.swing.*;
import java.awt.event.*;

class MouseEventDemo {
    JPanel panel;
    MousePressedReporter mousePressedReporter;
    JFrame frame;
    MouseEventDemo() {

        panel = new JPanel(); //1 create source − mouse click on panel generates event
        mousePressedReporter = new MousePressedReporter();//2c create listener

        panel.addMouseListener(mousePressedReporter);//3 register source with listener

        frame = new JFrame();
        frame.setSize(200, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        frame.add(panel);//panel added to frame

            frame.setVisible(true);//frame made visible, start of event driven execution
    }

     public static void main(String[] args) {
         new MouseEventDemo();
     }
}
```

43.3 Execution model

To incorporate events, the execution model is extended with a rule how to obtain the snapshots after an event has been generated.

Here the extension is made for mouse events only: . . . both indicate where the other mouse event handlers of the example besides `mouseClicked` and `mousePressed` should be added as well as where further extensions will be made later.

The first stage in generating and processing evens is taken care of by the *system*, i.e., does not involve Java code but rather rather issues like processing mouse actions on the screen area. As the result of a device action on a screen area corresponding to a component, like a mouse press on a `JPanel`, the system generates events, here a `mouseEvent`, an object with as data which component was involved as source and that it is an event of the kind mousePressed. Reactions to events takes time, so the system provides buffering by putting the events in a queue, the `eventQueue`.

From this point on, the execution model describes how the events, stored in the `EventQueue`, are processed, finally leading to the call of the corresponding handlers. This processing is implemented as Java library code. The execution of this code can be visualized using the execution model as is: there are no new programming concepts involved.

If library code is involved, the execution model usually provides snapshots that correspond to the API description of, e.g., a library method. In case of the event mechanism, the AOI discription is not detailed enough to give a good insight in the workings of the library code. In such cases, an alternative is to not use the API description, but visualize the execution of the actual library code.

The code is rather complex, so we provide a somewhat abstracted version of it that is easier to understand and sufficient to provide the snapshots for the execution model. The abstracted code consists of a class `SwingThread` that deals with how the event mechanism Where applicable, available Java methods are used; also simplified helper-methods of our choice are used the meaning of which is obvious. For example, the method `dispatchEvent` is actually present in all sources, whereas the method `isMouseListenerEvent` is a method, with obvious meaning, that we introduce to model a step in a `dispatchEvent` method. In the snapshots in the visualization control moves through the code we have written or instantiated, according to this abstracted code.

Note, that at various points information from the events (event are objects!) is used. Notably in the following cases.

1. There may be several components that act as event sources - `e.getSource()` provides the source of an event.
2. There may be several listeners of several types, e.g., `MouseListener` attached to a source - hence a source contains a list of listeners for each type - e.g., `isMouseListenerEvent(AWTEvent e)` provides the information which kind of event obtains (determining which type of listener the event should be processed by), and `List<MouseListener>mouseListeners` is an example of such list.
3. There are several handlers present in each listener - e.g., `isMouseClickedEvent(AWTEvent)e` provides the information which one matches the event.

The abstracted code is then as follows.

A new object is added to the visualization, of the following class.

```
class SwingThread {
  EventQueue<AWT event> eventQueue;//filled by the system
  AWTEvent e;

  void run {
    while (true) {
      if (!eventQueue.isEmpty()) {
        e = eventQueue.getNextEvent();//take topmost event from eventQueue
        e.getSource().dispatchEvent(e);//on component that is source, call dispatchEvent
      }
    }
}
```

For each source that is a (sub)class (of) `JComponent`, lists of references to listeners and a `dispatchEvent` method are provided as follows.

```
class JComponent {
  List<MouseListener> mouseListeners;//references to listeners registered on component,
     ...//one list for each listener type

  void dispatchEvent(Event e) {
    if (isMouseListenerEvent(e)) {//for e MouseListenerEvent,
      for (MouseListener lis : mouseListeners) {//on each MouseListener registered on component
        if (isMouseClickedEvent(e)) {//depending on type of e,
          lis.mouseClicked(e);//call handler matching type of e
        } else if (is MousePressedEvet(e)){
          lis.mousePressedEvent(e);
        } else if
           ...
    } else if
      ...
    }

}
```

For the example program this leads to the following visualization.

***Thread object `swingThread` with run added to objects, control moving from run to dispatch of event-source/event-matching listener/event-matching handler ***

43.4 Remarks

43.4.1 More mouse events

In addition to the group of `MouseEvent`s of the example, corresponding to the `MouseListener`, there is a group of `MouseEvent`s, corresponding to the `MouseMotionListener`. There is also another class of events for the mouse, `MouseWheelEvent`, with corresponding listener `MouseWheelListener`. See the API for details.

In the execution model, in the `JComponent` method `eventDispatch` clauses are added for the two extra listeners and their handlers.

43.4.2 Key events

Like the mouse, the keyboard keys is a device. The approach is quite similar: the counterparts for the `MouseListener` and `addMouseListener` are `KeyListener` and `addKeyListener`, respectively.

The difference is that unlike the mouse, nothing to click on is needed. However, something has to be provided to produce the events. Again `JComponent` is used for this purpose. As the component is not clicked on, it has to be identified as the source for the keys in a different way. During execution at most one of the sources always has the so called *focus*. The keystrokes have as source the source that has the focus; let this be named `KeySource`. If, e.g., as the result of a mouse action on another source than the one intended for the keys, then in the handler of the mouse event, `keySource.requestFocus();` should be written to return the focus to `KeySource`. In this case `KeySource` should explicitly allow the focus to be directed to it: this is done through `KeySource.setFocussable();`. See the API for details.

In the execution model, in the `JComponent` method `eventDispatch` clauses are added for the extra listener and its handlers.

The effect of the focus methods is reflected in variables of the codekeyEvent and the source.

43.4.3 Information in events

Apart from the information in the event object needed for the processing by the event mechanism, namely what the source of the event is and what the kind of event is, the event contains more information. This information can be used in programming the reaction to the event, i.e., in programming the handler.

What information can be obtained from an event of certain type can be found in the API.

For `mouseEvent`s, e.g., relevant information is the position on the panel where the mouse is used: `int getX()` returns the horizontal x position of the event relative to the source component. Similar for `getY()`.

In the context of the example, this information can be used to program a reaction that depend on where the mouse press occurred.

For `keyEvent`s, e.g., the most relevant information is which key was pressed: `public int getKeyCode()` returns the integer keyCode associated with the key in this event. Note, that the return values are also provided as constants: `static int VK_UP` is the constant for the non-numpad up arrow key. Similar for the down, left and right keys.

Chapter 44

Time driven programming – Timers

Sometimes program are required to show dynamic behavior governed by their own clock.

44.1 Aim

We want to bring time into the execution of a program in a controlled manner, i.e., to enable activity by the program to occur at controlled intervals, for example to create animations.

44.2 Means

We use a *timer object* in the program that triggers activity at controlled intervals

Objects instantiated from the class `Timer` in the package `javax.swing` generate action events at regular intervals. Code reacts to these events in the same way as in the case of `Button`: we implement the interface `ActionListener` and write the reaction in the body of the method `actionPerformed`.

The following example explains the most important features. The visible behavior is governed by the two `Timer` objects: `fastTicker` and `slowTicker`, which generate events at different rates. A circle moves horizontally across the screen at the rate of `fastTicker`. A number is displayed that decreases at the rate of `slowTicker`. Note that drawing occurs so often that the effect of each tick of `fastTicker` is made visible. If the method `this.repaint` in the example would be replaced by a more time consuming method, problems might arise.

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

class Animation extends JPanel implements ActionListener {
    int xpos = 0;
    int ypos = 0;
    int size = 12;
    int dx = 2; // stepsize for movement

    int number = 10;
    Timer fastTicker;
    Timer slowTicker;

    Animation() {
        fastTicker = new Timer(50, this);
        slowTicker = new Timer(1000, this);
    }

    void start() {
      fastTicker.start();
      slowTicker.start();
    }

    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == fastTicker) {
            xpos += dx;
        } else if (e.getSource() == slowTicker) {
            number −= 1;
        } else {
            System.err.println(”Action evt of unknown source”);
        }
        if (xpos > this.getWidth()−size || xpos < 0) {
            dx = −dx; // invert direction of movement
        }
        if (number <= 0) {
            slowTicker.stop();
        }
        this.repaint();
    }

    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        g.fillOval(xpos, ypos, size, size);
        g.drawString(String.valueOf(number), 100, 100);
    }
}

public class AnimationDemo {
        JFrame frame;
        Animation animation;

    void demo() {
        frame = new JFrame(”Animation Demo”);
        animation = new Animation();

        frame.getContentPane().add(animation);

        frame.setSize(200, 200);
        frame.setVisible(true);
        animation.start();
    }

   // main instantiates and uses the object
   public static void main(String[] args) {
        new AnimationDemo().demo();
   }
}
```

We have two Timer objects here that tick with different rates. A Timer object is created with the command

```
new Timer(time, listener())
```

where `time` is the tick interval in milliseconds and `listener` is an object whose class implements the

`ActionListener` interface. So a Timer is always created with at least one listener. Of course, you can add more listeners with the command `slowTicker.addActionListener(otherListener)`.

When created, a timer object is dormant. It starts ticking, i.e. generating events, when you call its start method. In our example:

```
slowTicker.start();
```

It can be stopped with the command

```
slowTicker.stop();
```

Note, that the position of the circle and the value of the number are recorded in instance variables. This makes it possible that the change that one method makes (the method `actionPerformed`) is "seen" by another method (the method `paintComponent`). This is an important technique in GUI programming and OO programming in general.

Consider the statement `g.drawString(String.valueOf(number), 100, 100);`. There, we want to write the value of `number` on the screen. Therefore, we first transform the value to a `String` by means of the method `String.valueOf()`.

Other methods

In the Java API you can find more information about the `Timer` class. (Note: there is also a `Timer` class in the package `java.util`; this class is used for other purposes.)

44.3 Execution model

As before.

Chapter 45

Basic graphics – paintComponent and Graphics object

We introduce drawing.

45.1 Aim

Produce drawings output (as opposed to textual output).

45.2 Means

- We define what should be drawn using the drawing methods from the graphics object `g`.
- We let the drawing be put on the screen by using inheritance, namely by overwriting the `paintComponent` method of the component in which we want the drawing to appear with the above mentioned definition of what should be drawn.

Out of the box, a `JFrame` or a `JPanel` doesn't do much more by itself than holding other components. If we want to draw shapes on them, we have to extend their behavior. How the inside of a component looks like is determined by the method `paintComponent(Graphics g)`. In the class `JPanel`, this method simply fills the inside of the panel with the background color. In a subclass of `JPanel`, this method can be overridden to do something more interesting. The parameter `Graphics g` is a reference to a helper object that controls the way things are drawn (colors, fonts, etc.). It contains methods to draw shapes and text, such as `drawOval`, `drawString`, and several more.

45.2.1 Callback

In Java, the programmer should not give drawing commands directly to the components. The reasons for this are (i) there are events outside the programmer's control that imply that a component should be (re)drawn: when a window is moved by the user from behind another window, etc. (ii) the time consuming task of drawing on the screen can be optimized when the Java system has more control over when exactly a part of the screen is drawn.

Therefore, the Java system uses an indirect way of drawing. When the system decides it is time to draw a component, it calls the method `paintComponent` with the appropriate `Graphics`-parameter. Every component has such a method. Drawing behavior should be defined by overriding this method. This concept of indirect control is called *callback*, since the program doesn't call the system, but the system calls the program.

When the programmer knows that a component needs to be redrawn (usually because he wants to show a change), he calls the method `repaint()` (without parameters) on that component. This will result in a *request* to the system to redraw the component as soon as possible. Consequently, the method `paintComponent` will be called. So you as a programmer will never call `paintComponent` yourself.

There is some sort of exception to this rule. When you start drawing inside a component, you usually want to start with a "blank slate". This is achieved by calling the `paintComponent` method *defined in the superclass*. Since the super class is the original class, it has no specific behavior; calling `paintComponent` only clears the inside. In Java, calling a method of the superclass is done by putting the keyword `super` in front of the call. You can see this as extending the behavior of the method with new behavior. Simply overriding the method would do away with the behavior of the superclass. By first calling the superclass's method the original behavior is incorporated.

45.2.2 Coordinates

Drawing commands take arguments about the size and location of the shapes. These are measured in integer numbers corresponding to pixels on the screen. The coordinate system has its origin, point $(0, 0)$, at the top left corner of the component that you are drawing on. The x-coordinates go from left to right, y-coordinates go *from top to bottom*. This is different from the mathematical convention, where y goes from bottom to top, but it is common in computer graphics.

The following example shows how to use `paintComponent`to paint an oval on the screen.

```java
//displays an oval on a JPanel
import javax.swing.*;
import java.awt.*;

class GraphicsPanel extends JPanel {

    public void paintComponent(Graphics g) {
        super.paintComponent(g);

        g.drawOval(100, 50, 100, 40);//implementation of drawing
    }
}

public class GraphicsDemo {
    JFrame frame;
    GraphicsPanel graphicsPanel;

    GraphicsDemo() {
        frame = new JFrame("GraphicsDemo");
        graphicsPanel = new GraphicsPanel();

        frame.add(graphicsPanel);

        frame.setSize(400, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }

    public static void main(String[] args) {
        new GraphicsDemo();
    }
}
```

We now extend the program to one that shows the use of `repaint` to increase the size of the oval as a reaction to a button click.

```
//reacts on mouse clicks by enlarging oval
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
// import java.swing.event.*;

class GraphicsPanel extends JPanel {
        int n;

        GraphicsPanel() {
                n = 1;
        }

        void nplus() {
                n = n+1;
        }

    public void paintComponent(Graphics g) {
        super.paintComponent(g);

        g.drawOval(100, 50, 100*n, 40*n);//implementation of drawing
    }
}

class MouseClickReporter implements MouseListener {//2a definition of listener as MouseListener

    GraphicsPanel graphicsPanel; //mouseClickReporter needs to reference graphicsPanel to increase value of n!

    void setGraphicsPanel(GraphicsPanel g) {
        graphicsPanel = g;
    }

    // this method will be called when the mouse is clicked
    public void mouseClicked(MouseEvent e) {
            graphicsPanel.nplus();//2b implementation of reaction by listener
            graphicsPanel.repaint();
    }

    public void mousePressed(MouseEvent e) {};//all unused abstract mouse event reactions
    public void mouseReleased(MouseEvent e) {};//that MouseListener offers
    public void mouseEntered(MouseEvent e) {};//have to be implemented too:
    public void mouseExited(MouseEvent e) {};//here just with dummy bodies {}
}

class MouseEventsPainting {
    JFrame frame;
    GraphicsPanel graphicsPanel;
    MouseClickReporter mouseClickReporter;

    MouseEventsPainting() {
        frame = new JFrame();//1 create source - mouse click on frame generates event
        graphicsPanel = new GraphicsPanel();
        mouseClickReporter = new MouseClickReporter();//2c create listener

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(400, 200);

        frame.add(graphicsPanel);

        frame.addMouseListener(mouseClickReporter);//3 register source with listener

        mouseClickReporter.setGraphicsPanel(graphicsPanel);

            frame.setVisible(true);//frame made visible, start of event driven execution
    }

     public static void main(String[] args) {
         new MouseEventsPainting();
     }
}
```

In the above program, it was awkward that the `MouseClickreporter` needed a reference to the the `GraphicsPanel`. We now change the structure, combining these two classes into one, `PaintReporter` that avoids this.

```
//reacts on mouse clicks by enlarging oval − panel and listener combined in one class
import javax.swing.∗;
import java.awt.∗;
import java.awt.event.∗;
// import java.swing.event.∗;

class PaintReporter extends JPanel implements MouseListener {//2a definition of listener as MouseListener
        int n;

        PaintReporter() {
                n = 1;
        }

     public void paintComponent(Graphics g) {
         super.paintComponent(g);

         g.drawOval(100, 50, 100∗n, 40∗n);//implementation of drawing
     }

   // this method will be called when the mouse is clicked
    public void mouseClicked(MouseEvent e) {
            n=n+1;//2b implementation of reaction by listener
            this.repaint();//NB repaint now has to be done on this!
    }

    public void mousePressed(MouseEvent e) {};//all unused abstract mouse event reactions
    public void mouseReleased(MouseEvent e) {};//that MouseListener offers
    public void mouseEntered(MouseEvent e) {};//have to be implemented too:
    public void mouseExited(MouseEvent e) {};//here just with dummy bodies {}
}


class MouseEventsPainting2 {
    JFrame frame;
    PaintReporter paintReporter;

    MouseEventsPainting2() {
        frame = new JFrame();//1 create source − mouse click on frame generates event
        paintReporter = new PaintReporter();//2c create listener

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(400, 200);

        frame.add(paintReporter);
        frame.addMouseListener(paintReporter);//3 register source with listener NB Still necessary!

            frame.setVisible(true);//frame made visible, start of event driven execution
    }

    public static void main(String[] args) {
        new MouseEventsPainting2();
    }
}
```

45.2.3 Some Graphics classes

class	description	some methods / fields / constructors	
Graphics	intermediate object used for drawing on components; contains drawing methods and some drawing data (pen color, font, etc.)	drawString drawImage drawLine draw/fillOval draw/fillRectangle	draw/fillPolygon draw/fillArc setColor setFont translate

45.3 Execution model

A sketch of the extension of the model.

Compare to the execution model for user event: now the system, possibly triggered by the `repaint` command, puts a repaint event in the events buffer to activate `paintComponent`.

Chapter 46

Basic graphics – paintComponent and Graphics object

Up till now, all interaction with the program was through console (or file) I/O. Now this is extended with graphics, displaying things on the screen.

46.1 Aim

Produce drawings on the screen as output (as opposed to textual output).

46.2 Means

We use classes from the Java API to make screen things.

There are two sets of GUI classes in Java: the basic Abstract Window Toolkit, in packages starting with `java.awt`, and the more versatile Swing classes, in packages starting with `javax.swing`. The Swing classes have more or less replaced the AWT classes and we will treat Swing in this book. Nevertheless, AWT classes are still available to support older programs. The names Swing classes start with a `J`, to distinguish them from their AWT counterparts. Some AWT classes, such as *Color* and the event classes are not replaced in Swing.

There are three kinds of GUI classes.

Component classes The things you see on the screen are all built from classes that are subclasses of the class *Component*. Therefore, we will call them components. (We will encounter more of these later.)

Container classes These classes, which are also components, can contain other GUI components. For example *JFrame*, a window, and *JPanel*, a rectangualr section of a window that may contain other components.

Helper classes These are not components and are not directly visible on the screen. They provide additional functionality for the GUI. For example *Color*.

To be able to draw, first something to draw on has to be provided. The following program creates a window and puts a blue panel on it.

```
//displays a frame and colored panel
import java.awt.*;
import javax.swing.*;

public class FrameDemo {
        JFrame frame;
        JPanel panel;
    Color color;

   FrameDemo() {
       frame = new JFrame ("example with component, container, and helper ");
                                                        // creates window: JFrame
       panel = new JPanel(); // creates component

       frame.add(panel); // puts panel in frame

       color = new Color(0, 100, 200); // creates helper object: color blue
       panel.setBackground(color); // colors background panel

       frame.setSize(400, 300);
       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                                                        // closing window exits program
       frame.setVisible(true); // makes graphis visible

   }

   public static void main(String[] args) {
       new FrameDemo();
   }
}
```

The position on screen and the size of components are measured in *pixels*. Many methods have two parameters to determine position or size. The first parameters always refers to the *horizontal* size or position, the second parameter refers to the *vertical size or position*. E.g., `frame.setSize(120, 100)`, sets the size of the window to 120 wide and 100 pixels tall.

Drawing is programmed as follows.

- We let the drawing be put on the screen by using inheritance, namely by overwriting the `paintComponent` method of the component in which we want the drawing to appear with the definition of what should be drawn.
- We define what should be drawn using the drawing methods from the graphics object `g`.

Out of the box, a `JFrame` or a `JPanel` doesn't do much more by itself than holding other components. If we want to draw shapes on them, we have to extend their behavior. How the inside of a component looks like is determined by the method `paintComponent(Graphics g)`. In the class `JPanel`, this method simply fills the inside of the panel with the background color. In a subclass of `JPanel`, this method can be overridden to do something more interesting. The parameter `Graphics g` is a reference to a helper object that controls the way things are drawn (colors, fonts, etc.). It contains methods to draw shapes and text, such as `drawOval`, `drawString`, and several more.

46.2.1 Callback

In Java, the programmer should not give drawing commands directly to the components. The reasons for this are (i) there are events outside the programmer's control that imply that a component should be (re)drawn: when a window is moved by the user from behind another window, etc. (ii) the time consuming task of drawing on the screen can be optimized when the Java system has more control over when exactly a part of the screen is drawn.

Therefore, the Java system uses an indirect way of drawing. When the system decides it is time to draw a component, it calls the method `paintComponent` with the appropriate `Graphics`-parameter. Every component has such a method. Drawing behavior should be defined by overriding this method. This concept of indirect control is called *callback*, since the program doesn't call the system, but the system calls the program.

When the programmer knows that a component needs to be redrawn (usually because he wants to show a change), he calls the method `repaint()` (without parameters) on that component. This will result in a *request* to the system to redraw the component as soon as possible. Consequently, the method `paintComponent` will be called. So you as a programmer will never call `paintComponent` yourself.

There is some sort of exception to this rule. When you start drawing inside a component, you usually want to start with a "blank slate". This is achieved by calling the `paintComponent` method *defined in the superclass*. Since the super class is the original class, it has no specific behavior; calling `paintComponent` only clears the inside. In Java, calling a method of the superclass is done by putting the keyword `super` in front of the call. You can see this as extending the behavior of the method with new behavior. Simply overriding the method would do away with the behavior of the superclass. By first calling the superclass's method the original behavior is incorporated.

46.2.2 Coordinates

Drawing commands take arguments about the size and location of the shapes. These are measured in integer numbers corresponding to pixels on the screen. The coordinate system has its origin, point $(0, 0)$, at the top left corner of the component that you are drawing on. The x-coordinates go from left to right, y-coordinates go *from top to bottom*. This is different from the mathematical convention, where y goes from bottom to top, but it is common in computer graphics.

The following example shows how to use `paintComponent`to paint an oval on the screen.

```
//displays an oval on a JPanel
import javax.swing.*;
import java.awt.*;

class GraphicsPanel extends JPanel {

    public void paintComponent(Graphics g) {
        super.paintComponent(g);

        g.drawOval(100, 50, 100, 40);//implementation of drawing
    }
}

public class GraphicsDemo {
    JFrame frame;
    GraphicsPanel graphicsPanel;

    GraphicsDemo() {
        frame = new JFrame("GraphicsDemo");
        graphicsPanel = new GraphicsPanel();

        frame.add(graphicsPanel);

        frame.setSize(400, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }

    public static void main(String[] args) {
        new GraphicsDemo();
    }
}
```

46.2.3 Some Graphics classes

class	description	some methods / fields / constructors	
Graphics	intermediate object used for drawing on components; contains drawing methods and some drawing data (pen color, font, etc.)	drawString	draw/fillPolygon
		drawImage	draw/fillArc
		drawLine	setColor
		draw/fillOval	setFont
		draw/fillRectangle	translate

46.3 Execution model

A sketch of the extension of the model.

The system, possibly triggered by the `repaint` command, puts a repaint event in a buffer, the *events buffer*, to activate `paintComponent`.