

# Interactivity and User Interfaces

## Chapter 3

## Content

- The AWT Event Model
- Keyboard Input
- Mouse Input
- Mouselook-Style Mouse Movement
- Creating an Input Manager
- Using the Input Manager
- Designing Intuitive User Interfaces
- Using Swing Components
- Creating a Simple Menu
- Letting the Player Configure the Keyboard
- Summary

2

## Introduction

- Interactivity is essential
- We will learn how to receive input (Keyboard, Mouse) from the user
- You will integrate this into your game and learn to implement UI with Swing
- Java SDK 1.4 and above does not allow joysticks
- Sources: we'll be organizing the reusable code into subpackages of `com.brackeen.javagamebook` (e.g. Chapter 2, "2D Graphics and Animation," are in the `com.brackeen.javagamebook.graphics` package)
- Finally, we'll need a simple class to make the quick test programs easier to implement. The **GameCore** class does just that, implementing some of the common techniques from the previous chapter, such as setting the display mode and running an animation loop. All we have to do is extend this abstract class and implement the **draw()** and **update()** methods.

3

## Introduction (2)

- By default, the **update()** method doesn't do anything, but in subclasses, you'll use it for updating sprites and such. Also, you'll probably extend the **init()** method to do things such as load images or initialize any variables.

4

## AWT Event Model

- Where does the AWT dispatch these events? When an event occurs on a particular component, the AWT checks to see if there are any listeners for that event.
- A *listener* is an object that receives events from another object. In this case, events come from the AWT event dispatch thread.
- There is a different type of listener for every type of event. For example, for key input events, there is a **KeyListener** interface.

5

## AWT Event Model (2)

- Here's an example of how the event model works for a key press:
  1. The user presses a key.
  2. The operating system sends the key event to the Java runtime engine.
  3. The Java runtime engine posts the event to the AWT's event queue.
  4. The AWT event dispatch thread dispatches the event to any **KeyListener**s.
  5. The **KeyListener** receives the key event and does whatever it wants with it.
- All listeners are interfaces, so any object can be a listener by implementing a listener interface. Also note that there can be several listeners for the same event type.

6

## AWT Event Model (3)

- There is a way to capture all AWT events. Although doing this isn't useful for a real game, it can be helpful for debugging your code or seeing what events get dispatched.

- Creating an `AWTEventListener`:

```
Toolkit.getDefaultToolkit().addAWTEventListener(  
    new AWTEventListener() {  
        public void eventDispatched(AWTEvent event) {  
            System.out.println(event);  
        }  
    }, -1);
```

- Remember, don't use something like this in a real game; use it only for debugging.

7

## Keyboard Input

- For a game, you want to use lots of keys, such as the arrow keys for movement and maybe the Control key for firing a weapon. We really aren't going to deal with things like text input—we can leave that for Swing components discussed later.

- To capture key events, you need to do two things:

1. Create a **KeyListener** and
2. Register the listener to receive events. To register a listener, just call the **`addKeyListener()`** method on the component that you want to receive key events on. For games here, that component is the full-screen window:

```
Window window = screen.getFullScreenWindow();  
window.addKeyListener(keyListener);
```

8

## Keyboard Input (2)

- To create a `KeyListener`, you just need to create an object that implements the `KeyListener` interface. The `KeyListener` interface has three methods:
  - **`keyPressed()`**, **`keyReleased()`**, and **`keyTyped()`**.
  - The "typed" event occurs when a key is first pressed and then repeatedly based on the key repeat rate.
- Each of these three methods takes a **`KeyEvent`** as a parameter. The `KeyEvent` object enables you to inspect what key was actually pressed or released, in the form of a virtual key code.
- A virtual key code is the Java-defined code to a particular keyboard key.

9

## Keyboard Input (3)

- All the virtual key codes are defined in `KeyEvent` in the form **`VK_XXX`**.
- For example, the Q key has the key code **`KeyEvent.VK_Q`**. Most of the time, you can guess the name of a key code (such as `VK_ENTER` or `VK_I`), but be sure to look up the `KeyEvent` class in the Java API documentation for the full list of virtual key codes.

See source code `keytest.java`

10

## Keyboard Input (4)

- Side effect to `KeyTest`
  - Useful program for testing how keys behave on different systems. Not the same!
  - Example: *key repetition*. When the user holds down a key, the operating system sends multiple events for that key.
    - ▶ In a text editor, for example, when you hold down the Q key, several Qs appear.
    - ▶ Linux: both the key press and the key release event are sent for each key repeat.
    - ▶ Windows: only the key press event is sent for each repetition, and the key release event isn't sent until the user actually releases the key.
- Also, there could be other subtle differences, such as the Windows Start key behaving differently depending on what version of Windows is running. Or, the key behavior could be different depending on what Java VM is running.

11

## Mouse Input

- You can receive three different types of mouse events:
  - Mouse button clicks
  - Mouse motion
  - Mouse wheel scrolls
- Mouse buttons behave like keyboard buttons, but without the key repetition. Mouse motion is broken down into x and y screen coordinates. Finally, mouse wheel events tell how far the wheel was scrolled.
- Each mouse event type has its own listener: **`MouseListener`**, **`MouseMotionListener`**, and **`MouseWheelListener`**. Each listener takes a **`MouseEvent`** as a parameter.

12

## Mouse Input (2)

- **MouseListener** interface has methods for detecting mouse presses, releases, and clicks (pressing and then releasing). We'll ignore clicks in our games just as we ignore KeyTyped events and deal with just presses and releases. `getButton()` method of `MouseEvent` for telling which button was pressed or released.
- Also, the **MouseListener** interface has methods to signal when the mouse has entered and exited the component. Because the component we use covers the entire screen, we can ignore these methods as well.
- For mouse movement, you can detect two types of motion with the `MouseMotionListener` interface:
  - Regular motion
  - Drag motion
- `getX()` and `getY()` of `MouseEvent` for current position

13

## Mouse Input (3)

```
void mouseMoved(MouseEvent e) {
    Point p = new Point(e.getX(), e.getY());
    trailList.addFirst(p);
    while (trailList.size() > TRAIL_SIZE)
        trailList.removeLast();
}
```

- The **MouseWheelListener** uses a subclass of `MouseEvent` called **MouseWheelEvent**. It has the method `getWheelRotation()` to inspect how much the mouse wheel was moved. Negative values mean scrolling up, and positive values mean scrolling down.

See source code `MouseTest.java`

14

## Mouselook-Style Mouse Movement

- The Java API doesn't directly provide methods to detect the relative motion of the mouse, but there are ways to trick the mouse to do what you want.
- To do this, you just need to make sure the mouse never hits the edge of the screen. Every time the mouse moves, you'll just reposition the mouse to the center of the screen. That way, the mouse will never be stopped by the edge of the screen, and you can always calculate how much the mouse has moved based on its previous location.
- Here's a breakdown of how it works:
  1. The mouse starts at the center of the screen.
  2. The user moves the mouse, and you calculate how much it moved.
  3. You send an event to reposition the mouse to the center of the screen.
  4. When you detect that the mouse was re-centered, you ignore the event.

15

## Mouselook-Style Mouse Movement (2)

- You can reposition the mouse using the **Robot** class. The `Robot` class was designed to automate testing of graphical applications. Besides being capable of programmatically moving the mouse, it has all sorts of functions for doing things such as emulating key presses and making screen captures. Moving the mouse is as simple as you'd like it to be:

```
robot.mouseMove(x, y);
```

See source code `MouselookTest.java`

16

## Hiding the Cursor

- Luckily, the Java API has methods to change the mouse cursor. Unluckily, the Java API doesn't define an invisible cursor.
- Some of the cursors it does define are listed here:

<code>CROSSHAIR_CURSOR</code>	A cursor in the shape of a plus sign
<code>DEFAULT_CURSOR</code>	The normal arrow cursor
<code>HAND_CURSOR</code>	The hand cursor that you normally see when you mouse over hyperlinks on web pages
<code>TEXT_CURSOR</code>	The text cursor, normally I-shaped
<code>WAIT_CURSOR</code>	The wait cursor, normally an hourglass

17

## Hiding the Cursor (2)

- The Java API enables you to create your own cursors using custom images, so you'll just create a cursor that has a blank image. You do this by calling the `createCustomCursor()` method of the `Toolkit` class:

```
Cursor invisibleCursor =
    Toolkit.getDefaultToolkit().createCustomCursor(
        Toolkit.getDefaultToolkit().getImage(""),
        new Point(0,0),
        "invisible");
```

- Here you just create an "invalid" image, which the `Toolkit` interprets as being an invisible cursor. You can change the cursor for your games by calling the `setCursor()` method:

```
Window window = screen.getFullScreenWindow();
window.setCursor(invisibleCursor);
```

- Later, you can get the default cursor by calling `Cursor's getPredefinedCursor()` method

18

## Creating an Input Manager

- One thing you might have noticed are the synchronized methods. Remember that all events are sent in from the AWT event dispatch thread, which is a different thread than the main thread.
- For future games, we handle all input at a specific point in the game loop. You can do this easily by setting Boolean variables when a key is pressed.
  - Example: a **jumpsPressed** Boolean can be set in the **keyPressed()** method when the spacebar is pressed, and later in the game loop, you can check whether the **jumpsPressed** variable is set. If it is, the code can make the player jump.
- Also, for some actions, such as jumping, you want to perform an action only on the initial key press. For other actions, such as moving, you want to move as long as the key is down.
- For jumping, if you want to detect initial key presses for jumping, whenever we see that **jumpsPressed** is true in the game loop, you can just set it to false. That way, the player won't jump again until he or she presses the key again.

19

## Creating an Input Manager (2)

- To do this, you need to be able to map a generic action to a key and let the action be mapped to different keys on the fly. For now, put this on the wish list for your **InputManager** class.
- To sum it up, here's everything you want **InputManager** to do:
  1. Handle all key and mouse events, including relative mouse movement.
  2. Save the events so you can query them precisely when you want to, instead of changing game state in the AWT event dispatch thread.
  3. Detect the initial press for some keys and detect whether the key is held down for others.
  4. Map keys to generic game actions, such as mapping the spacebar to a jump action.
  5. Change the key mapping on the fly so the user can configure the keyboard.

20

## Creating an Input Manager (3)

- Also, it has code for mapping keys and mouse events to **GameActions**. When a key is pressed, the code checks to see whether a **GameAction** is mapped to that key and, if so, calls the **GameAction**'s **press()** method.
- How does the mapping work? An array of **GameActions** is created. Each index in the array corresponds to a virtual key code. Most of the virtual key codes have a value that is less than 600, so the array of **GameActions** has a length of 600.
- The mapping works similarly for mouse events. Because mouse events don't have codes, fake mouse codes are made up in **InputManager**. There are mouse codes for mouse movement (left, right, up, and down), mouse wheel scrolling, and mouse buttons.

See source code `GameAction.java`

See source code `InputManager.java`

21

## Creating an Input Manager (4)

- One additional feature: functionality to get the name of the key or mouse event.
- Use **getKeyName()** and **getMouseName()** methods, useful for telling the user which keys do what (configuration!).
- Also, the **InputManager** has the **resetAllGameActions()** method, which resets any **GameActions** that are currently "pressed." This can be useful for situations such as switching from the main menu to the game. Without clearing the **GameActions**, pressing the spacebar in a menu could cause the player to jump later, as soon as the game starts.

22

## Using the Input Manager (Pause)

- Pausing the game, two things happen:
  - Object and animations aren't updated
  - Input is ignored (except for the key to unpause the game)
- Modify animation loop

```
if (!paused) {
    checkInput();
    updateGameObjects();
}
```
- Of course, you'll continue drawing the screen even if the game is paused. You could do extra drawing as well, such as having an animated "paused" message. Also, you'll be sure to update the paused state when the user presses the **P** key, for example.

23

## Using the Input Manager (Gravity)

- Gravity
  - Effect of gravity is essentially a downward acceleration.
  - In **Sprite** class from the last chapter, the sprite's position is updated based on its velocity. Likewise, for a sprite with gravity, the sprite's velocity is updated based on its acceleration. For example:

```
velocityY = velocityY + GRAVITY * elapsedTime;
```
- Gravity of the Earth is about 9.8 meters per second, but that's not important for your game.
- You're using pixels and milliseconds as your measurements, so you'll just use whatever value "feels" right. Here, the value of **GRAVITY** is 0.002. This allows the player to jump but prevents the player from jumping past the top of the screen.

See source code `Player.java`,  
`InputManagerTest.java`

24

## Designing Intuitive User Interfaces

- Designing a useful, intuitive, and attractive user interface is one of the most important aspects of creating a game. Lacking a useful interface can make a game no fun to play.
- User Interface Design Tips. Basic rule is to keep it simple, descriptive, and fast:
  - Keep your user interfaces simple and uncluttered. Not every option has to be presented to the user at once. Instead, you can keep the most common or most useful options on the main screen and provide an easy way to view the less common options.
  - Make sure every option or button is easy to get to. If it takes too many clicks to find certain functionality, it will frustrate the users.

25

## Designing Intuitive User Interfaces (2)

- User Interface Design Tips (cont.):
  - Use Tooltips. Tooltips are pop-up descriptions that appear when your mouse lingers over a particular object. They can tell you what a button does or give status on other user interface elements. Tooltips are a quick way to answer the question "What's this?" Swing has Tooltips built in, so it's easy to implement.
  - Give the user a response to every action. This can be as simple as playing a sound or displaying the wait cursor. Also avoid pauses between the time a user clicks something and when the action occurs.
  - Test your user interfaces on someone you know. What a button does might be obvious to you, but to someone else it could be confusing. When you test to see how someone else might use your interface, don't say anything—just watch to see what the person does and take notes. Remember, in the real world, when users play your game, you're not going to be right next to them telling them what to do!

26

## Designing Intuitive User Interfaces (3)

- User Interface Design Tips (cont.):
  - After some people test your user interfaces, ask them what they think would be easier or more useful. Also ask them whether the icons make sense. Don't hold them back or say things like, "But the code won't work that way." Just listen and take everything into consideration. Who knows, later you might find a way to make the code "work that way" after all.
  - Be prepared to overhaul your user interface if it just doesn't work. You might have spent days coding and creating icons for what you thought was the perfect interface, but don't fret if you have to throw it all away. If it's to create a better game, it's worth it.

27

## Using Swing

- Swing is a set of classes used to create user interface elements. It includes all sorts of common components such as windows, buttons, pop-up menus, drop-down lists, text-input boxes, check boxes, and labels.
- You'll use Swing components to implement your user interfaces. You've already seen Swing's **JFrame** used as the full-screen window in the **ScreenManager** class.
- Because Swing renders its own components, you can draw Swing components in your own rendering loop. This is great news because it means that you can integrate all of Swing's functionality into your full-screen games.
- Swing has a pretty large API, you can customize.

28

## Swing Basics

- Most user interface elements are a class, so you just have to create an instance of that class. For example, to create an OK button, simply create a new **JButton** object:

```
JButton okButton = new JButton("OK");
```

- This creates a **JButton** with all the default parameters and with a text label of OK. Swing components start with J, as in **JLabel**, **TextField**, and so on, and they're all derived from the **JComponent** class.
- The components are added to a top-level container, which, in this case, is the **JFrame**. Components are added to the **JFrame**'s content pane. When a component is added to the content pane, Swing handles all the mouse clicks for it, so you don't have to. Here's an example of adding a button to the content pane:

```
JFrame frame = screen.getFullScreenWindow();  
frame.getContentPane().add(okButton);  
okButton.setLocation(200, 100);
```

29

## Swing Basics (2)

- The second way to lay out components is to use a **LayoutManager**. **LayoutManagers** take each component's size into consideration and position the components according to certain rules. Several different **LayoutManagers** exist, and each positions components in different ways. **FlowLayout**, for example, lays out components on the screen in a left-to-right, top-to-bottom fashion.
- To use a **LayoutManager**, call the **setLayout()** method of the content pane. Here's an example of using a **FlowLayout**:

```
frame.getContentPane().setLayout(new FlowLayout());
```

30

## Creating a Simple Menu

- First, when you click a button, what exactly happens? Swing sees the click and checks to see whether the button has any **ActionListeners**. If it does, those ActionListeners are notified that the button was clicked in the AWT event dispatch thread.
- Like KeyListener or MouseListener, ActionListener is an interface that any object can implement. It has one method named **actionPerformed()** that takes an **ActionEvent** as a parameter. You can check to see what component generated the event by calling ActionEvent's **getSource()** method. For example:

```
public void actionPerformed(ActionEvent e) {
    Object src = e.getSource();
    if (src == okButton) {
        // do something
    }
}
```

31

## Creating a Simple Menu (2)

- Finally, in the user interface, you can do a few things to your buttons to make them more usable:
  - Add Tooltips. Just call something like `setToolTip("Hello World")`, and Swing handles the rest.
  - Use icons. Instead of having text in your buttons, you can use icons. There can be different icons for the default, rollover, and pressed states.
  - Hide the default look. You want your icons to appear by themselves, so turn off the button's border and call `setContentAreaFilled(false)` to make sure the button background isn't drawn.
  - Change the cursor. Make the cursor appear as a hand when you roll over a button by calling the `setCursor()` method.
  - Turn off key focus. If the button is focusable, the button can "steal" keyboard focus away from the game when it is clicked. To fix this, just call `setFocusable(false)`. The only drawback to this is that only the mouse can activate a button.

See source code `MenuTest.java`

32

## Keyboard Configuration

- The keyboard configuration feature can be broken into two parts:
  1. create the configuration dialog box
  2. create a special component that enables the user to enter a key or mouse click.

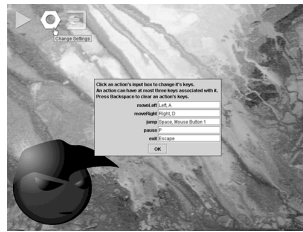
- The dialog box lists all the possible game actions for a game and also contains instructions and an OK button.

The dialog box itself is a JPanel, and everything within the dialog box can be created using a series of components, panels, and layouts.

- Creating the special input component is a little more difficult. You need this component to display what key is currently mapped to a game action, and also to enable the user to press a key or a mouse button to change its setting. When input is done, you'll also have to make sure that the input component gives the keyboard focus back to the game's main window.

See source code `KeyConfigTest.java` `InputComponent`

33



## Keyboard Configuration (2)

- InputComponent is a subclass of **JTextField**. Need to override JTextField's input methods.
- Every Swing component is an instance of the **Component** class. The Component class has methods such as `processKeyEvent()` and `processMouseEvent()`. Override KeyListener or MouseListener methods and enable input events by calling the `enableEvents()` method.
- Alternative way to listen for input events.

34