# events # group-project # gui

# Handling Key Events

**TCSS 305 Programming Practicum**

> **Demo Project**
>
> The code examples in this guide come from the demo project: TCSS305-Game

Keyboard input is what turns a GUI application into something you can actually *play*. Moving a piece with WASD, pausing with Escape, rotating with the arrow keys--all of these depend on detecting and responding to what the user does with the keyboard. This guide covers Java's key event system: the interface that defines three event methods, the adapter class that keeps your code clean, three progressively better patterns for mapping keys to actions, and the focus gotcha that silently breaks keyboard handling for nearly everyone the first time.

> **Related Guides**
>
> - This guide assumes familiarity with event-driven concepts. For the theory behind events, listeners, and the event loop, see the Event-Driven Programming guide.
> - For mouse event handling (the sibling to this guide), see the Handling Mouse Events guide.

---

## 1 The KeyListener Interface

Java provides the `KeyListener` interface for receiving keyboard events. Unlike mouse events (which are split across two interfaces), all keyboard events live in a single interface with three methods:

| Method | Fires When |
| --- | --- |
| `keyPressed(KeyEvent e)` | A key is pressed down |
| `keyReleased(KeyEvent e)` | A key is released |

| Method | Fires When |
| --- | --- |
| `keyTyped(KeyEvent e)` | A key press produces a character (Unicode input) |

That's three abstract methods in one interface.

## 1.1 keyPressed vs. keyTyped vs. keyReleased

The distinction between these three methods matters more than it first appears:

- **`keyPressed`** fires the moment a physical key goes down. It fires for *every* key, including non-character keys like Shift, Ctrl, arrow keys, and Escape. If the user holds a key down, `keyPressed` fires repeatedly (key repeat). **This is the method you want for game controls.**
- **`keyReleased`** fires when the physical key comes back up. Useful for detecting when the user *stops* holding a key--for example, stopping a character's movement when the player lifts their finger.
- **`keyTyped`** fires when a key press produces a Unicode character. It does *not* fire for non-character keys (arrow keys, function keys, Shift alone, Ctrl alone). The `keyCode` field is always `VK_UNDEFINED` in a `keyTyped` event--you use `e.getKeyChar()` instead. **This is the method you want for text input.**

## 1.2 The Event Lifecycle

When the user presses and releases a character key:

```
keyPressed  ->  keyTyped  ->  keyReleased
```

When the user presses and releases a non-character key (like an arrow key):

```
keyPressed  ->  keyReleased
```

No `keyTyped` event fires for non-character keys because no character was produced.

When the user holds a key down (key repeat):

```
keyPressed  ->  keyTyped  ->  keyPressed  ->  keyTyped  ->  ...  ->
keyReleased
```

The `keyPressed` and `keyTyped` events repeat, but `keyReleased` fires only once at the end.

**For Game Controls, Use keyPressed**

Game controls need immediate, repeating response. `keyPressed` fires for all keys (including arrows and WASD), fires repeatedly when held, and provides the key code via `e.getKeyCode()`. This is why virtually every Java game uses `keyPressed` for movement controls.

## 1.3 Working with KeyEvent

Every key event handler receives a `KeyEvent` parameter. The most important methods:

| Method | Returns | Use Case |
|---|---|---|
| `e.getKeyCode()` | `int` | The virtual key code (`KeyEvent.VK_W`, `KeyEvent.VK_UP`, etc.) -- use in `keyPressed` / `keyReleased` |
| `e.getKeyChar()` | `char` | The Unicode character produced -- use in `keyTyped` |
| `e.isShiftDown()` | `boolean` | Whether Shift was held during the event |
| `e.isControlDown()` | `boolean` | Whether Ctrl was held during the event |
| `e.isAltDown()` | `boolean` | Whether Alt was held during the event |

**getKeyCode() vs. getKeyChar()**

In `keyPressed` and `keyReleased`, use `e.getKeyCode()` to identify which key was pressed. In `keyTyped`, use `e.getKeyChar()` to get the character produced. Using `getKeyCode()` in a `keyTyped` handler always returns `KeyEvent.VK_UNDEFINED`. Using `getKeyChar()` in a `keyPressed` handler may return unexpected results for non-character keys.

---

# 2 Is This a Functional Interface? No!

If you've worked with `ActionListener`, you might try writing a key handler as a lambda:

```
// This does NOT compile
panel.addKeyListener(e -> System.out.println("Key pressed!"));  // ERROR
```

`KeyListener` has **three** abstract methods. It is not a functional interface. A lambda expression can only implement a single abstract method, so there's no way for the compiler to know which of the three methods your lambda is supposed to represent.

> **Important**
>
> A functional interface has **exactly one** abstract method. `KeyListener` has three. It cannot be used with lambda expressions. You need a class that implements all of the interface's methods.

Implementing `KeyListener` directly means providing bodies for all three methods, even if two of them do nothing:

```
// Tedious: implementing all three methods just to handle keyPressed
public class MyListener implements KeyListener {
    @Override
    public void keyPressed(KeyEvent e) {
        // The one method we actually care about
        System.out.println("Pressed: " + KeyEvent.getKeyText(e.getKeyCode()));
    }

    @Override
    public void keyReleased(KeyEvent e) { }  // Empty

    @Override
    public void keyTyped(KeyEvent e) { }  // Empty
}
```

That's familiar boilerplate. The same problem--and the same solution--as with `MouseListener`.

> **Related Guides**
>
> - For the `ActionListener` pattern where lambdas *do* work, see the Adding Event Handlers guide.
> - For the parallel discussion of `MouseListener` and `MouseAdapter`, see the Handling Mouse Events guide.

---

## 3 KeyAdapter to the Rescue

`KeyAdapter` is an abstract class that implements `KeyListener` with empty method bodies. You extend it and override only the methods you care about--the same pattern as `MouseAdapter`.

```
// Clean: only override what you need
KeyAdapter adapter = new KeyAdapter() {
    @Override
    public void keyPressed(KeyEvent e) {
        System.out.println("Pressed: " + KeyEvent.getKeyText(e.getKeyCode()));
    }
};
```

No empty `keyReleased` or `keyTyped` bodies. Just the behavior you need.

### 3.1 Using KeyAdapter as an Inner Class

In practice, you'll define your adapter as a private inner class:

```
public class GameController extends JPanel {

    private final GameControls myGame;

    public GameController(GameControls theGame) {
        myGame = theGame;
        addKeyListener(new MyKeyHandler());
        setFocusable(true);
    }

    private class MyKeyHandler extends KeyAdapter {

        @Override
        public void keyPressed(KeyEvent e) {
            // Handle keyboard input
        }
    }
}
```

Notice the `setFocusable(true)` call in the constructor. We'll explain why that's critical in Section 5.

> **Tip**
>
> Using a named inner class (rather than an anonymous class) is preferable when your adapter contains non-trivial logic like key mapping. It keeps your constructor clean and gives the handler a meaningful name.

---

## 4 Three Patterns for Key Mapping

Once you have a `KeyAdapter` handling `keyPressed`, you need to decide *how* to map key codes to actions. This is where the interesting design decisions live. We'll look at three patterns, progressing from simplest to most elegant.

For all three patterns, assume this setup:

```
// A game model with movement methods
private final GameControls myGame;
```

### 4.1 Pattern 1: The if-else Chain

The most obvious approach--check each key code with conditional logic:

```
@Override
public void keyPressed(KeyEvent e) {
    final int keyCode = e.getKeyCode();

    if (keyCode == KeyEvent.VK_W) {
        myGame.moveUp();
    } else if (keyCode == KeyEvent.VK_S) {
        myGame.moveDown();
    } else if (keyCode == KeyEvent.VK_A) {
        myGame.moveLeft();
    } else if (keyCode == KeyEvent.VK_D) {
        myGame.moveRight();
    }
}
```

**Strengths:** Simple, easy to understand, works. Every student who has completed Programming 1 can read this.

**Weaknesses:**

- **Verbose** -- each key requires 2-3 lines of branching logic

- **Hard to maintain** -- adding a new key means inserting another `else if` branch into the middle of a growing chain
- **Mixes concerns** -- the mapping logic (which key maps to which action) is interleaved with the dispatching logic (executing the action)

For a game with 4 movement keys, this is manageable. For a game with 15+ key bindings, the `keyPressed` method becomes a wall of if-else branches.

## 4.2 Pattern 2: Map-Based Lookup

Instead of conditional branches, store the key-to-action mapping in a `Map<Integer, Runnable>`:

```
private final Map<Integer, Runnable> myKeyMap = Map.of(
    KeyEvent.VK_W, myGame::moveUp,
    KeyEvent.VK_S, myGame::moveDown,
    KeyEvent.VK_A, myGame::moveLeft,
    KeyEvent.VK_D, myGame::moveRight
);

@Override
public void keyPressed(KeyEvent e) {
    myKeyMap.getOrDefault(e.getKeyCode(), () -> { }).run();
}
```

**What's happening:** The `Map` stores each key code as a key and a `Runnable` (a method reference to the game action) as the value. When a key is pressed, we look up the key code in the map. If it exists, we run the associated action. If not, `getOrDefault` returns a no-op lambda `() -> { }` that does nothing.

**Strengths:**

- **Concise** -- the entire `keyPressed` method is one line
- **Declarative** -- the map reads like a configuration table: "W means up, S means down, A means left, D means right"
- **Easy to extend** -- adding a new key binding means adding one entry to the map, not modifying the dispatching logic
- **Separation of concerns** -- the mapping is data; the dispatching is code

**Weaknesses:**

- Slightly more abstract than the if-else chain--students seeing `Map<Integer, Runnable>` for the first time may need a moment to understand the pattern

**Why Runnable?**

`Runnable` is a functional interface with a single method: `void run()`. It's the perfect fit for "an action that takes no arguments and returns nothing"--exactly what a key binding does. Method references like `myGame::moveUp` implement `Runnable` because `moveUp()` matches the `void run()` signature.

## 4.3 Pattern 3: Switch Expression

Java's enhanced switch expressions provide a clean, readable way to map key codes to actions:

```
@Override
public void keyPressed(KeyEvent e) {
    mapKeys(e.getKeyCode()).run();
}

private Runnable mapKeys(int theKeyCode) {
    return switch (theKeyCode) {
        case KeyEvent.VK_W -> myGame::moveUp;
        case KeyEvent.VK_S -> myGame::moveDown;
        case KeyEvent.VK_A -> myGame::moveLeft;
        case KeyEvent.VK_D -> myGame::moveRight;
        default -> () -> { };
    };
}
```

**What's happening:** The `mapKeys` method uses a switch *expression* (not a switch *statement*) that returns a `Runnable`. Each `case` maps a key code to a method reference. The `default` case returns a no-op lambda. The `keyPressed` method calls `mapKeys` to get the appropriate action and then `.run()` s it.

This is the pattern used in our demo project's `GameController` class.

**Strengths:**

- **Clean separation** -- the mapping logic lives in its own method
- **Readable** -- the switch expression reads naturally as "for this key, do this action"
- **Exhaustive** -- the `default` case ensures every key code is handled (no silent gaps)
- **Returns a value** -- switch expressions are expressions, not statements, so they can return directly

**Weaknesses:**

- Requires familiarity with enhanced switch syntax (Java 14+)

### 4.4 Comparing the Three Patterns

| Pattern | Lines per Key | Readability | Extensibility | Best For |
|---|---|---|---|---|
| if-else chain | 2-3 | Immediate | Low (modify method body) | Prototyping, very few keys |
| Map-based lookup | 1 | High (declarative) | High (add map entry) | Many key bindings, configurable controls |
| Switch expression | 1 | High (structured) | Moderate (add case) | Clean separation, moderate key count |

All three patterns produce identical behavior. The choice is about code organization and maintainability.

**Gen AI & Learning: Describing Key Mappings to AI Tools**

When asking an AI coding assistant to add keyboard controls, describe the mapping as data: "W moves up, S moves down, A moves left, D moves right." This framing naturally leads to the Map or switch patterns. If you say "when the user presses W, call moveUp; when they press S, call moveDown..." you're more likely to get an if-else chain. The way you describe the problem shapes the solution you get--from both human collaborators and AI assistants.

---

## 5 The Gotcha: Focus and KeyListener

This is the single most common mistake when working with `KeyListener`, and it's the keyboard equivalent of the mouse event dual-registration gotcha.

> **The Focus Requirement**
>
> A `KeyListener` only receives events when the component it's registered on **has keyboard focus**. If your component doesn't have focus, `keyPressed` will never fire--silently. No error, no warning, just nothing.

## 5.1 The Problem

Here's the scenario. You create a panel with a `KeyAdapter`, register it, run the program--and nothing happens when you press keys:

```
public class GamePanel extends JPanel {

    public GamePanel() {
        addKeyListener(new KeyAdapter() {
            @Override
            public void keyPressed(KeyEvent e) {
                System.out.println("Key pressed!");  // Never fires!
            }
        });
    }
}
```

You press keys. Nothing. You check your method signature, add print statements, question your life choices. Everything looks correct, but no key events are received.

## 5.2 Why It Happens

Swing's keyboard event system is tied to **focus**. At any given moment, exactly one component in the application has keyboard focus, and that component (and only that component) receives key events. Think of it like a conversation: only the person you're talking to hears what you say.

By default, a `JPanel` is not focusable. It never receives focus, so it never receives key events. Even if you click on the panel, it won't grab focus because Swing doesn't consider it a component that should accept keyboard input.

This is different from mouse events, which fire on any component the cursor is over regardless of focus. Mouse events are position-based; key events are focus-based.

## 5.3 The Fix

Two things are needed:

1. **Make the component focusable** by calling `setFocusable(true)`
2. **Request focus** so the component actually receives it

```
public class GamePanel extends JPanel {

    public GamePanel() {
        addKeyListener(new KeyAdapter() {
            @Override
            public void keyPressed(KeyEvent e) {
                System.out.println("Key pressed!");  // Now it fires!
            }
        });

        setFocusable(true);           // Allow this component to receive focus
        requestFocusInWindow();       // Ask for focus now
    }
}
```

`setFocusable(true)` tells Swing that this component *can* receive keyboard focus. `requestFocusInWindow()` asks Swing to give it focus immediately.

> **Focus Can Be Stolen**
>
> If your panel contains buttons or other focusable components, clicking a button will move focus to the button and away from the panel. Your `KeyListener` will stop receiving events until the panel regains focus.
>
> This is why the demo project's button handlers call `requestFocusInWindow()` after each button click:
>
> ```
> myUpButton.addActionListener(theEvent -> {
>     myGame.moveUp();
>     requestFocusInWindow();  // Return focus to the controller
> panel
> });
> ```
>
> Without this call, pressing a button would break keyboard controls until the user clicks on the panel again.

> **Tip**
>
> Make it a habit: whenever you add a `KeyListener` to a component, immediately add `setFocusable(true)` in the same constructor. If you override any `keyPressed` method and it doesn't fire, focus is the first thing to check.

---

## 6 A Note on Key Bindings

Everything in this guide uses `KeyListener` and `KeyAdapter`, which is the right starting point for understanding keyboard events. But you should know that Swing provides a more powerful alternative: **Key Bindings** via `InputMap` and `ActionMap`.

Key bindings solve several problems that `KeyListener` cannot:

- **No focus issues** -- key bindings can be configured to work even when the component doesn't have focus
- **Changeable key assignments** -- remapping a key is a single `put()` call on the `InputMap`, no code changes in the handler
- **Cleaner separation** -- input mapping and action execution are completely decoupled

```
// Preview of key bindings (you'll use this in a later sprint)
InputMap inputMap = getInputMap(WHEN_IN_FOCUSED_WINDOW);
ActionMap actionMap = getActionMap();

inputMap.put(KeyStroke.getKeyStroke("W"), "moveUp");
actionMap.put("moveUp", new AbstractAction() {
    @Override
    public void actionPerformed(ActionEvent e) {
        myGame.moveUp();
    }
});
```

> **You'll Use This Later**
>
> Key bindings are the recommended approach for production Swing applications. You'll encounter `InputMap` / `ActionMap` in a later sprint when you implement changeable key bindings. For now, `KeyAdapter` with the patterns from Section 4 is the right tool for learning keyboard event handling.

---

## 7 In Our Demo Codebase

The demo project's `GameController` class demonstrates the keyboard patterns discussed in this guide.

The controller uses a `KeyAdapter` inner class with a switch expression to map WASD keys to game actions. Notice that `keyPressed` checks whether the player is stunned before dispatching--when the NPC collides with the player, input is blocked for several ticks:

```
private final class MyControlsKeyAdapter extends KeyAdapter {
    @Override
    public void keyPressed(final KeyEvent theEvent) {
        if (!isStunned()) {
            mapKeys(theEvent.getKeyCode()).run();
        }
    }

    private Runnable mapKeys(final int theKeyCode) {
        final Runnable doNothing = () -> { };

        return switch (theKeyCode) {
            case KeyEvent.VK_W -> myGame::moveUp;
            case KeyEvent.VK_S -> myGame::moveDown;
            case KeyEvent.VK_A -> myGame::moveLeft;
            case KeyEvent.VK_D -> myGame::moveRight;
            default -> doNothing;
        };
    }
}
```

Key design decisions in this code:

- `KeyAdapter` rather than `KeyListener` -- only `keyPressed` is needed, so the other two methods are inherited as no-ops
- **Switch expression** returns a `Runnable` -- clean separation between key mapping and action execution
- **Method references** ( `myGame::moveUp` ) -- concise and readable
- `doNothing` **variable** for the default case -- slightly more readable than an inline `() -> { }`
- **State gating** -- the `isStunned()` check prevents all key input during the stun period, keeping the guard logic separate from the mapping logic
- **Focus management** -- the button handlers call `requestFocusInWindow()` to return focus to the controller panel after each click

This is Pattern 3 from Section 4 in action.

---

## Summary

| Concept | Key Point |
| --- | --- |
| **KeyListener** | Handles keyboard events: keyPressed, keyReleased, keyTyped (3 methods) |

| Concept | Key Point |
| --- | --- |
| **keyPressed vs. keyTyped** | `keyPressed` fires for all keys and repeats when held--use for game controls. `keyTyped` fires only for character-producing keys--use for text input |
| **Not a Functional Interface** | `KeyListener` has three abstract methods--no lambdas allowed |
| **KeyAdapter** | Abstract class implementing `KeyListener` with empty methods--override only what you need |
| **if-else Chain** | Simplest key mapping pattern; works but verbose and hard to maintain |
| **Map-Based Lookup** | Declarative key mapping; easy to extend, single-line dispatch |
| **Switch Expression** | Clean, structured key mapping; returns a value, used in demo project |
| **Focus Requirement** | `KeyListener` only works on focused components--call `setFocusable(true)` and `requestFocusInWindow()` |
| **Focus Stealing** | Clicking a button moves focus away from your panel--call `requestFocusInWindow()` in button handlers |
| **Key Bindings** | `InputMap` / `ActionMap` is the more robust alternative--you'll use it in a later sprint |

## Further Reading

**External Resources**

- Oracle Java Tutorial: How to Write a Key Listener - Official guide to handling key events
- Oracle Java Tutorial: How to Use Key Bindings - Official guide to InputMap/ActionMap
- Baeldung: Java KeyListener - Practical tutorial with examples

# References

**Primary Texts:**


- Horstmann, C. S. (2022). *Core Java, Volume I: Fundamentals* (12th ed.). Oracle Press. Chapter 11: Event Handling -- Key Events.
- Bloch, J. (2018). *Effective Java* (3rd ed.). Addison-Wesley. Item 42: Prefer lambdas to anonymous classes.


**Language Documentation:**

- Oracle JDK 25: `KeyListener` -- Interface for receiving keyboard events
- Oracle JDK 25: `KeyAdapter` -- Abstract adapter class for receiving key events
- Oracle JDK 25: `KeyEvent` -- Event object indicating a key action occurred
- Oracle JDK 25: `InputMap` -- Maps KeyStrokes to action names
- Oracle JDK 25: `ActionMap` -- Maps action names to Actions

**Tutorials:**

- Oracle Java Tutorial: How to Write a Key Listener -- Official Swing key event tutorial
- Oracle Java Tutorial: How to Use Key Bindings -- Official Swing key bindings tutorial

---

*This guide is part of TCSS 305 Programming Practicum, School of Engineering and Technology, University of Washington Tacoma.*