CS 360
Programming Languages
Event-Driven Programming

Events and Timers and Listeners, Oh My!

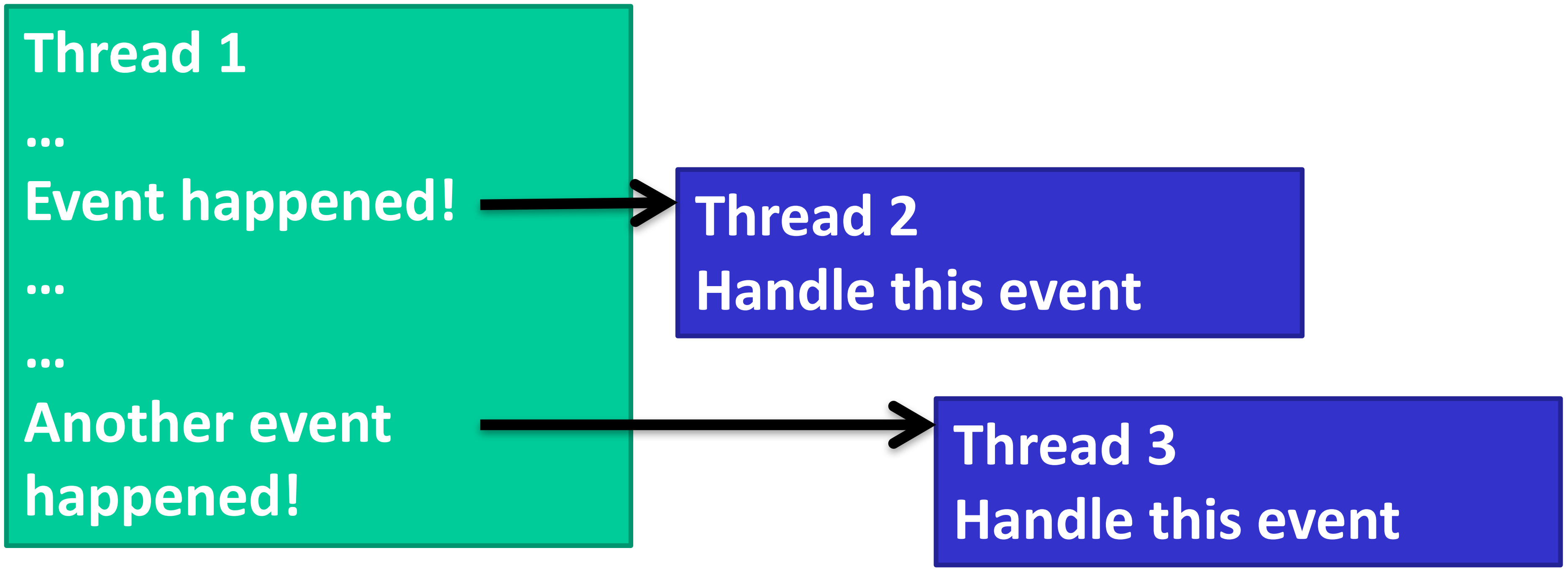
Control flow

- "Traditional" program: one statement at a time, line by line.

- Threaded program: CPU determines execution order.
  - Controlled with `synchronized`, `wait()/notifyAll().`

- Event-driven program: controlled by the order that "events" happen.

- Event-driven programming is often seen in threaded programs, as another model of communication between threads.



- An ***event*** is something that happens in your program that another piece of code wants to be aware of.

  - Simple things: mouse clicks, key presses, …
  - Complex things: file is done loading, calculation is finished, received request from a client.

- Event-driven programming is no better or worse than other models of thread communication, it's just different.

  - Often forced to use it because so many graphical user interface (GUI) libraries use it.

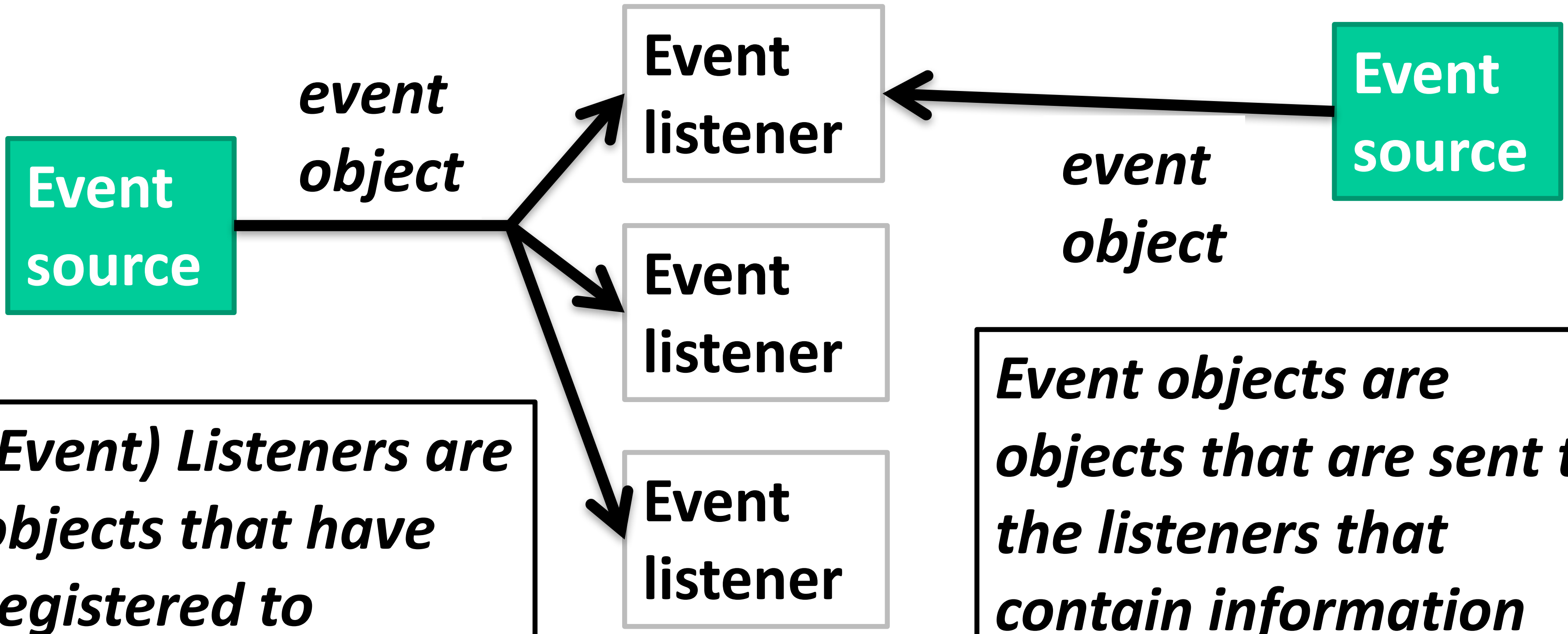
Here's the way Java does it:

- Java has certain classes that generate events (**sources**).
  - Usually classes that correspond to visual elements on the screen: buttons, menus, etc.

- Programmers write other classes that are called **event listeners**.
  - These classes have certain methods that will be automatically called in response to events.

- Programmers link up an event generator (a source) with an event listener.
  - Extra information is sent from the source to the listener through **event objects.**

- ***Sources***, ***event objects***, and ***listeners***.



(Event) Listeners are objects that have registered to receive certain types of events from event sources.

Event objects are objects that are sent to the listeners that contain information about the event that occurred (e.g., where the mouse was clicked).

Let's look at an example...

- Look at the `EventExample.java` code.

- `JButton`: a class that models a button.
  - Also an event source.

- `HelloWorldListener`: a class designed to listen for button presses.
  - The code that runs when the action happens (inside actionPerformed) is called an ***event handler***.

- `ActionEvent` (arg type to `actionPerformed`) is the event class.
  - Whenever the `JButton` is pushed, it triggers (fires) an `ActionEvent`.
  - Has methods for determining which object caused the event, when it happened, etc.

- Connected through `addActionListener` function.

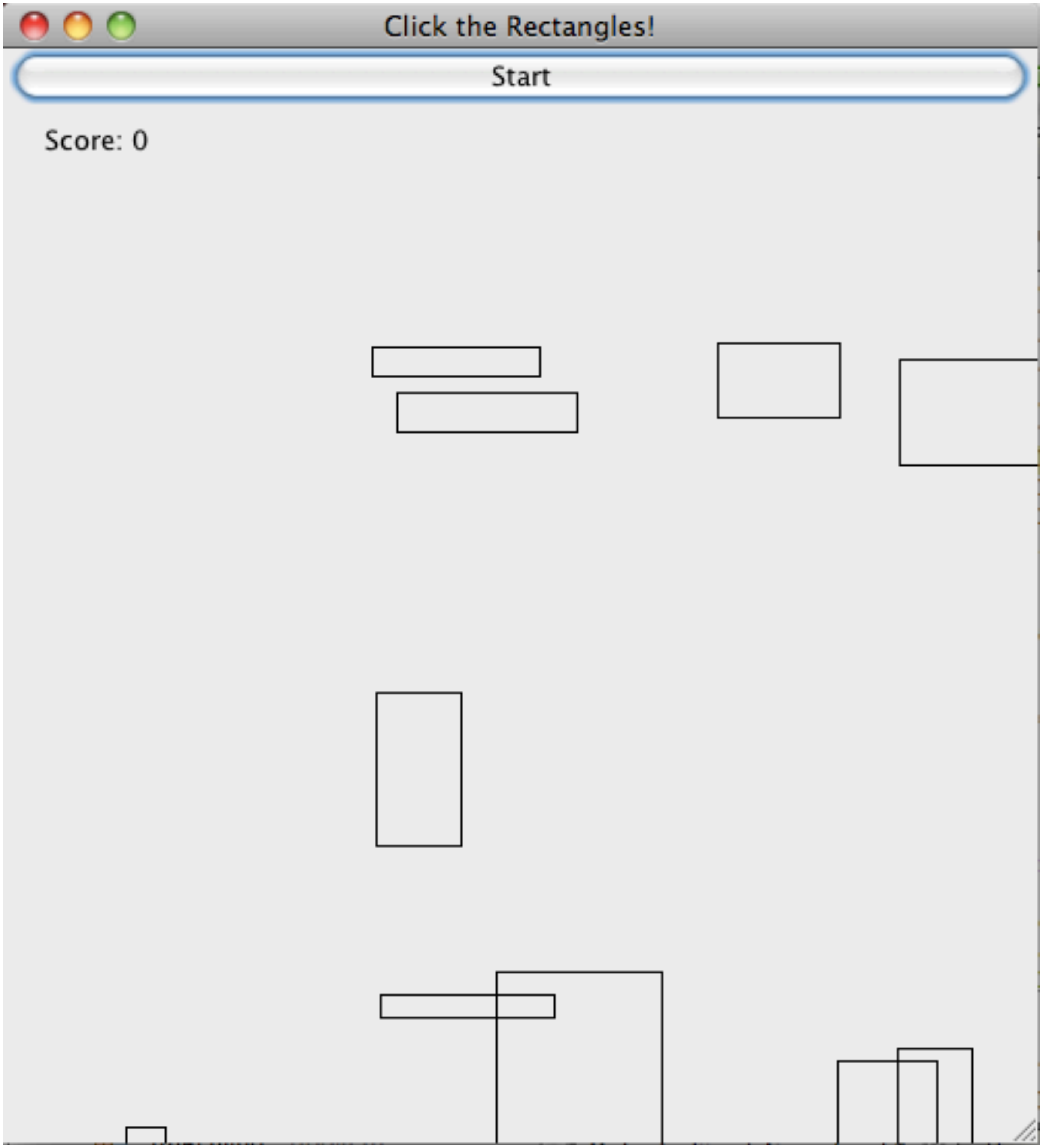
- Purpose of events: separate the code that **causes** the event from the code that **handles** the event.

- Lets one event source trigger multiple actions
  - `JButton` can have multiple listeners added.

- Lets one listener listen to multiple event sources.
  - Could have `HelloWorldListener` connected to many buttons, key presses, drop-down menus, etc.

- Java has (many) classes for Events:
  - `ActionEvent`, `MouseEvent`, `KeyEvent`, …
- and classes for Listeners:
  - `ActionListener`, `MouseListener`, `KeyListener`, …

- We're going to examine just buttons and the mouse today.


Click the Rectangles!
Start
Score: 0

- From class website, get ClickRectangleStart.java.
  - Paste into new NetBeans project.

- GameFrame: represents the window that holds the game.
  - Contains a "panel" to hold the moving rectangles, and a JButton to start the game.

- GamePanel: represents the moving rectangles area.
  - moveShapesToLeft: moves all rectangles to the right.
  - handleMouseClick: event handler for when the panel is clicked.
  - paintComponent: draws the rectangles on the screen.

Run It

Task 1: Start Button

- In StartButtonActionListener
  - Write actionPerformed.
  - This method should call gameArea.moveShapesToLeft().
  - Then call repaint() [tells Java to redraw the rectangles]

- Uncomment lines in the GameFrame constructor to attach the listener to the button.

- When done, you should be able to click the button and the shapes should move to the left one pixel per click.

Task 2: Mouse clicks

- In GameMouseClickListener:
  - Write mouseReleased.
  - This should call handleMouseClick.
    - arguments should be event.getX() and event.getY()
  - Call repaint()   [asks Java to redraw the rectangles]

- In the GameFrame constructor, uncomment lines to attach the listener to the mouse.

Task 3: Automatic scrolling

- We don't want to click the start button to advance the rectangles.

- We need a way to automatically fire events in rapid succession.
  - In order to repeatedly call moveShapes every few milliseconds to give the illusion of scrolling.

Solution: Timer

- Timer objects will fire an ActionEvent repeatedly every *x* milliseconds.
- Timer t = new Timer(x, <action listener>);
- t.start();

- [See TimerExample.java]

- In MoveShapesActionListener:
  - Write actionPerformed to do two things:
    - call moveShapesToLeft on gameArea
    - call repaint() [request that Java redraw the rectangles]

- Rewrite start button listener:
  - actionPerformed should do three things:
    - Create a new MoveShapesActionListener
    - Create a timer: args are 10 (milliseconds), and your move shapes action listener.
    - Start the timer.