

GUI Programming

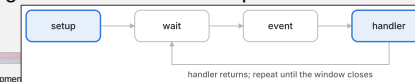
Event-Driven Programming

A major shift –

- programs so far have run top-to-bottom, starting in `main` and asking for input when they want it
 - the program controls the flow and dictates the order of interaction



- in GUI programs, most things happen only in response to user actions – *event-driven programming*
 - the user dictates the order of interaction
 - the program decides how to respond



CPSC 329: Software Development

2

Frameworks

Something has to –

- wait for the next event
- work out which code should respond to it
- call that code, then go back to waiting



Every GUI program needs the same functionality.

- written once and reused – a *framework*
- the framework provides the structure and control flow
 - JavaFX is a GUI framework
- you supply the parts specific to your application
 - e.g. what Klondike does when a card is clicked

CPSC 329: Software Development • Fall 2026

3

Inversion of Control

- with a library, your code is in charge of the flow of control
 - you call it when you need something – it does the work and returns, `Collections.sort`
 - e.g. `Math.max`, `ArrayList.add`, `Collections.sort`
- with a framework, the framework is in charge of the flow of control
 - it calls your code at points it defines
 - JUnit calls your `@Test` methods
 - JavaFX calls your `start()` and your event handlers
- *inversion of control* – the framework, not your code, owns the overall flow
 - don't call us, we'll call you
- many toolkits are both libraries and frameworks
 - e.g. JavaFX is used as a library when you create new components and call methods on them

4

The Event Loop

- the framework runs the event loop

```
while (running) {  
    Event e = eventQueue.take(); // wait until something happens  
    dispatch(e);                // find the handler(s) and call them  
}
```

- where events come from
 - the OS – mouse clicks, key presses, window resizes
 - timers – animations, clocks
 - the program itself – code can fire events too
- events are handled one at a time, in order*

Terminology

term	meaning	Klondike example (JavaFX)
event source	the object where the event happens	the Draw button
event	an object describing what happened	an ActionEvent
handler / listener	code to run when the event happens	"move top card from stock to waste"
registration	telling the event source about the handler	<code>drawButton.setOnAction(...)</code>
dispatcher	the framework code that calls the handler	inside JavaFX
adapter	a class with empty default handler methods (so you only have to override the ones you need)	n/a – <i>adapter</i> is an older term (from Swing)

Handlers are Callbacks

- a *callback* is code you hand to someone else to run later
- in `setOnAction(handler)`, *handler* is a behavior object
 - you decide what happens (by defining the behavior)
 - the framework decides when (by invoking the behavior)

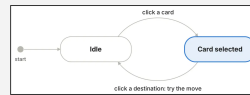
Where Event-Driven Programming Shows Up

Event-driven programming shows up when the program must react to an outside world that it can't schedule.

- desktop and mobile GUIs – taps, clicks, gestures
- web front ends – JavaScript `onClick`
- games – input events plus timer ticks
- servers – "a request arrived"
- robots and embedded systems – sensor readings, interrupts

Stateless vs Stateful Handlers

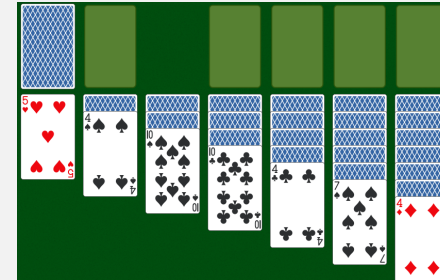
- in a *stateless* handler, the same event means the same response every time
 - e.g. New Game → deal a fresh game
 - e.g. Rules → show a rules dialog
- in a *stateful* handler, the response depends on what has happened before
 - e.g. click a tableau card
 - if nothing is selected → select it
 - if a card is already selected → try to move it on top of the card clicked
 - stateful handlers are modeled with state diagrams
 - the program must remember where the user is in the interaction along with any relevant info
 - e.g. which card, if any, is currently selected



9

Exercise

- sketch the state machine for move a card by clicking
- which controls should be enabled in each state?



CPSC 329: Software Development • Fall 2026

10

JavaFX

- JavaFX is Java's modern GUI toolkit
 - successor to Swing (the successor to AWT)
- has been separate from the JDK since Java 11
 - must download separately
 - Eclipse setup involves two pieces
 - where to find JavaFX at compile time – build path
 - where to find JavaFX at runtime – VM arguments

CPSC 329: Software Development • Fall 2026

11

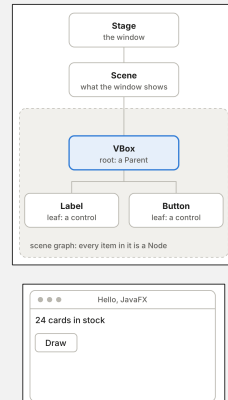
CPSC 329: Software Development • Fall 2026

12

Core Concepts

JavaFX uses a theater metaphor –

- the *stage* is the window
- the *scene* is what is showing in the window right now
- the *scene graph* is the tree of things in the scene
 - hierarchical structure
 - every item in the tree is a Node
 - layout panes (e.g. VBox) hold other nodes – Parents
 - in the tree you build, controls (e.g. Label, Button) are leaves



JavaFX Program Structure

What's the difference between the stage and the scene?

- the Stage object represents the window
 - it has methods like setTitle to manipulate the window
- the Scene object represents the contents of the window
 - it holds the scene graph which contains the elements (such as buttons) that are in the window

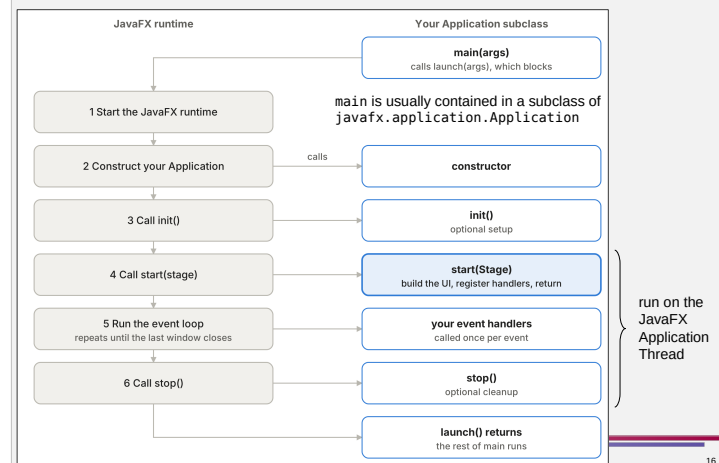
Do you have to have both a Stage and a Scene?

- yes – you can't have a GUI program without a window, and there's no point in a window with nothing in it

JavaFX Application Lifecycle

- main is usually contained in a subclass of javafx.application.Application
- main calls launch(args), then JavaFX takes over –
 - constructs an instance of your Application subclass
 - calls init() – hook; default does nothing
 - calls start(Stage) – you build the scene graph here
 - runs the event loop until the last window closes
 - calls stop() – hook; default does nothing
- handlers and start() run on the JavaFX Application Thread
 - remember that handlers are run one at a time, in order

JavaFX Application Lifecycle



JavaFX Program Structure

Is the start method always in the main class of the program?

- `public void start ( Stage stage )` throws Exception acts like the main program – it is where you set up the user interface
- main could be in another class, but there is little reason for it to be

A Minimal JavaFX Application

```
public class HelloFX extends Application {

    @Override
    public void start(Stage stage) {
        Label status = new Label("24 cards in stock");
        Button draw = new Button("Draw");
        draw.setOnAction(e -> { status.setText("You drew a card"); });

        VBox root = new VBox(10, status, draw);
        stage.setScene(new Scene(root, 300, 150));
        stage.setTitle("Hello, JavaFX");
        stage.show();
    }

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

Writing a Handler – Syntax Alternatives

- named class
 - reusable and can be tested on its own but verbose

```
class DrawHandler implements EventHandler<ActionEvent> {
    public void handle(ActionEvent e) { /* ... */ }
}
draw.setOnAction(new DrawHandler());
```
- anonymous inner class
 - avoids creating many small named classes and handler action is defined where it is registered but still verbose

```
draw.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent e) { /* ... */ }
});
```
- lambda (and method reference)
 - possible because `EventHandler` is a functional interface (it has exactly one abstract method)
 - method reference form is shorthand for a lambda that only calls an existing method

```
draw.setOnAction(e -> drawCard());
draw.setOnAction(this::onDraw); // void onDraw(ActionEvent e)
```

Draw Counter Example

Pitfalls

One thread runs the show –

- while a handler runs, the JavaFX Application Thread can't
 - process other clicks
 - redraw the window
 - respond to resize or close

Handlers must return quickly.

- slow work (searching, network access, loading files, ...) goes on another thread

Freeze Demo

Best Practices

- write handlers as short lambdas that call a well-named method
 - long handler bodies are hard to read and bury logic where it can't be named, reused, or tested

```
draw.setOnAction(new EventHandler<ActionEvent>() {  
    public void handle(ActionEvent e) { /* ... */ }  
});
```

Best Practices

- keep game/business logic out of handlers

 // Tangled: game rules live inside the UI

```
draw.setOnAction(e -> {  
    if (stock_.isEmpty()) {  
        while (!waste_.isEmpty()) stock_.push(waste_.pop());  
    } else {  
        waste_.push(stock_.pop());  
    }  
    status.setText(stock_.size() + " cards in stock");  
});
```

 // Separated: the handler translates, the game decides

```
draw.setOnAction(e -> {  
    game_.drawFromStock();  
    status.setText(game_.stockSize() + " cards in stock");  
});
```

- handler's job is to interpret the event → invoke the game/business logic → update what is on screen
 - invoke the logic, don't implement it
- why it matters
 - testability – can unit test drawFromStock()
 - design – separation of concerns (game class shouldn't know a UI exists)

Quick Check

```
Button undo = new Button("Undo");  
undo.setOnAction(e -> game_.undo());
```

- name the event source, the handler, and the registration step
- what type is the event object?
- why doesn't `start()` contain a loop that waits for clicks?