Graphical User Interfaces (GUIs)

• Large and important class of event-driven programs
  – waits for user-interaction events
  – mouse clicks, button presses, etc.

• Desktop, Mobile, Web, etc. provide libraries to write these
  – each of these use callbacks & events
  – examples of “event-driven programs”

• Using these libraries decreases bugs
  – also gives users a familiar experience
GUI Libraries

• Core parts of these applications:
  – stores some data for the user
  – displays that data for the user
  – allows the user to change the data
    • causes the app to re-display

• Early apps required a lot of code to implement these

• More recent improvements have made this easier
  – highly valuable
    • your time is important
  – less code (usually) means fewer bugs
GUI Libraries

- AWT & Swing are the native Java libraries for writing GUIs
  - Android apps are also GUIs and written in Java

- Core parts of these applications:
  - stores some data for the user
  - displays that data for the user
  - allows the user to change the data
    - causes the app to re-display

- Library helps with the latter two parts
  - components used to display data
  - components allow listeners that are notified of interaction
AWT / Swing Example

SimpleFieldDemo.java
Events in GUI Libraries

Most of the GUI widgets can generate events
  – button clicks, menu picks, key press, etc.

Add a **listener** to be called back when those events occur
  – component promises to call you in those circumstances
  – passed an **event** object that provides info about the event

More examples of “callbacks” coming later…
Achievement unlocked: Callbacks

**Callback pattern:** “Code” provided by client to be used by library
- In JS etc., pass a function as an argument
- In Java, pass an object with the “code” in a method

Examples: `HashMap` calls its client’s `hashCode`, `equals`

**Synchronous** callbacks:
- Useful when library needs the callback result immediately

**Asynchronous** callbacks:
- *Register* to indicate interest and where to call back
- Useful when the callback should be performed later, when some interesting event occurs
Event listeners / handlers

*Event listeners* must implement the proper interface. AWT/Swing:
- **KeyListener** – handle key press
- **ActionListener** – handle button press
- **MouseListener** – handle mouse clicks
- **MouseMotionListener** – handle mouse move/drag

When an event occurs
- the appropriate method specified in the interface is called:
  - `actionPerformed`, `keyPressed`, `mouseClicked`, ...
- an event object is passed to the listener method

Interfaces are different in Android but all conceptually the same
Event objects

GUI event is represented by an *event object*
- passes information often needed by the handler

In AWT/Swing, the superclass is `AWTEvent`. Some subclasses are:
  - `ActionEvent` – GUI-button press
  - `KeyEvent` – keyboard
  - `MouseEvent` – mouse move/drag/click/button

In Android, the superclass is `InputEvent`.

Event objects contain
- UI object that triggered the event
- other information depending on event. Examples:
  - `ActionEvent` – text string from a button
  - `MouseEvent` – mouse coordinates
Achievement unlocked: Observers

This is the observer pattern

- Objects can be observed via observers/listeners that are notified via callbacks when an event (of interest) occurs
- Pattern: Something used over-and-over in software, worth recognizing when appropriate and using common terms
- Widely used in public libraries

More examples of “observers” coming later…
GUI Client Programming

• Clients sit around waiting for events like:
  – mouse move/drag/click, button press, button release
  – keyboard: key press or release, sometimes with modifiers like shift/control/alt/etc.
  – finger tap or drag on a touchscreen
  – window resize/minimize/restore/close
  – timer interrupt (including animations)
  – network activity or file I/O (start, done, error)
    • (we will see an example of this shortly)
Event-driven programming

An *event-driven* program is designed to wait for events:

- program initializes then enters the *event loop*
- abstractly:
  ```
  do {
    e = getNextEvent();
    process event e;
  } while (e != quit);
  ```

Contrast with most programs we have written so far

- they perform specified steps in order and then exit
- that style is still used, just not as frequently
  - example: computing Page Rank or other Big Data work
UI Thread

- Where is the event loop in these Swing programs?

- The library creates a separate thread that runs that event loop
  - the “UI thread”
  - created when the JFrame is made visible
  - application does not exit until this thread also finishes
    - that happens automatically when the window is closed
Problems with SimpleFieldDemo

- Code is too **verbose**
  - can be improved using Lambda syntax

- Code is *not at all* **modular**
  - one file that mixes data, presentation, interaction

- **Too much work** involved with laying out elements
Easier Layout Idea #1: Just Say No

• Much of the difficulty here has to do with resizing…

• Do we really need to support resizing?

• Two platforms restrict resizing in some ways:
  – Android / iPhone
  – Bootstrap (HTML)
iPhone / Android Layout

- iPhone and iPad come in fixed sizes
- Just give a fixed layout for each possible size
Bootstrap (HTML)

• Width is restricted to one of 5 values (phone up to huge screen)
  – library automatically switches to best match for screen width
  – can use the same design for multiple sizes if you wish

• Still allows arbitrary height for the content
Bootstrap Example

BootstrapDemo.html
Easier Layout Idea #2: Declarative UI

• How much of layout needs to be code?
  – does this really require forward / backward reasoning?

• iPhone / Android show that this can be done
  – only for fixed sized screens

• HTML can be used as a more declarative language for UI
  – (.NET and other frameworks have comparable toolkits)
HTML

• **Hyper-Text Markup Language**

• Language for writing documents shown in a web browser
  – co-opted to display the UI for Web apps

• Document is a sequence of tags and text
Anatomy of a Tag

Element

<p> Some Text </p>

Tag Name  Content  Closing Tag
Anatomy of a Tag

\[
\text{Element} \quad \langle p \text{id="firstParagraph"} \rangle \text{ Some Text } \langle /p \rangle
\]

Tag Name  Attribute Name  Attribute Value  Content  Closing Tag
Tags form a Tree

<p>Some Text</p>
<br />
<div>
  <p>Hello</p>
</div>

This tree, as it lives in the browser, is often called the "DOM" – Document Object Model
A Few Useful Tags

• See the W3Schools HTML reference for a complete list, along with all their supported attributes.

• Some worth knowing:
  • `<p>` - Paragraph tag, surrounds text with whitespace/line breaks.
  • `<div>` - “The curly braces of HTML” - used for grouping other tags. Surrounds its content with whitespace/line breaks.
  • `<span>` - Like `<div>`, but no whitespace/line breaks.
  • `<br />` - Forces a new line (like “
”). Has no content.
  • `<html>` and `<head>` and `<body>` - Used to organize a basic HTML document.
HTML for UI

- Consists tags and their content
  - components become tags
    - input fields, buttons, etc.
    - e.g., `<button>`
  - containers have start and end tags
    - tags placed in between are children
    - e.g., `<div>` and `<p>`
    - additional information provided to the tag with “attributes”

- HTML removes the need for `panel.add` calls
  - parent / child relationship *implied* by tree structure
HTML + JS

• To make an app we also need **code**

• Code is provided inside a `<script>` tag
  – all browsers support the JavaScript language
  – more in a moment…
HTML + JS UI Example

HtmLFieldDemo.html
HTML + JS + CSS

- Cascading Style Sheets allow separation of styling from rest
  - *styling* is colors, margins, etc.
  - allows non-programmers to take some of this work
    - code produces document structure (tree of tags)
    - changes to tags require agreement by both parties
Dynamic Web Content

• Earlier example had a fixed set of components.
  – same for iPhone / Android apps

• More realistic apps need to change the set of components displayed on the screen dynamically
  – consider Gmail as an example
  – need the components to come from code
JS Example

register/index.js
Remaining Problems

- Code is extremely **verbose**
  - can be improved using Lambdas

- Code is *not sufficiently* **modular**
  - one JS mixes data, display, interaction

- **Too much work** involved with laying out elements

- **Poor tool support**
  - HTML is created in strings!
  - (and other issues not mentioned so far…)