CDER Center

- Began in 2010
- Developed PDC curriculum by 2012 that influenced ACM 2013, revised 2020
- Working with PD knowledge area group of ACM/IEEE 202X
- Helped ABET change accreditation criteria
- Conducted 24 PDC education workshops, multiple SIGCSE special sessions
- Early adopter and training/development grants **(one this summer)**
- Two books, journal special issues
- Guided AP Computer Science Principles to include PDC
- Work with other PDC education groups, e.g., CS in Parallel, SIGCSE

Introductions

- Who you are
- Where you are from
- What you teach
- What you hope to gain from the workshop

Introducing Concurrency Concepts in CS1

Sowing seeds for future growth

Why early PDC?

- Nearly all systems today use concurrency and distribution
- This is the computing model incoming students are familiar with
- This is model employers expect our graduates to understand
  - Employers report CS grads take 6 months to 2 years to onboard, before they are able to do useful work (in part due to a lack of PDC concepts)
  - Students report that they learn more from an internship than in the first two years of the curriculum, most of which wasn’t useful
- Where are we going wrong?

What Do We Currently Teach?

- Hello World
  - Sequence, console output
- Raw data types (int, float, double, char, etc.) and String
  - Everything has to be built up from machine primitives
  - You can’t do anything more complex until you understand this
- Variables and constants
  - Variables in programming are like variables in math - hold simple values
  - Values either change or they don’t - no middle ground or combination

What Do We Currently Teach?

- Assignment
  - If something is allowed to change, this is how we change it
- Types and casts
  - A value is a group of bits, whose form sometimes needs conversion
- Complex output is string concatenation and line formatting for printing
  - Everything is translated to a string for output
  - Importance of spacing
  - Need to understand end of line and buffer flushing

```
lastFirst = LAST + ", " + TITLE + " " + FIRST + ", ";
cout << "Name in last-first-initial format is ";
cout << lastFirst << MIDDLE << '.' << endl;
```

What Do We Currently Teach?

- Function calls
  - …are like functions in math
  - Give them a value or two, and they return a simple value
  - … or sometimes not, or sometimes they do and we ignore it
  - Complex code is arranged in functions
- Some data types come with a library of functions (like String)
  - Those functions may be applied with a different syntax (.)
  - But it’s easy to think of the identifier on the left as just another argument

What Do We Currently Teach?

- Input comes from characters that a user types on a line
  - We break the characters into groups to fit into variables
  - The characters are converted, depending on the type of the variable
  - If the program or the user messes up, it crashes
  - Important to tell the user what and how to type
- Think of input as a form of parsing
  - Read position and end of line are key concepts for input
- Files are the other kind of input
  - But they’re like frozen console input without prompts

A common pattern we teach:

```
cout << "Enter the part number:" << endl;              // Prompt
cin >> partNumber;
cout << "Enter the quantity of this part ordered:"     // Prompt
     << endl;
cin >> quantity;
```

A common way to visualize input:

Statements		Contents After Input	Marker Position in the Input Stream
1.			25 A 16.9\n
	cin >> i;	i = 25	25 A 16.9\n
	cin >> ch;	ch = 'A'	25 A 16.9\n
	cin >> x;	x = 16.9	25 A 16.9\n

What Do We Currently Teach?

- Branches are how we check console input for errors
- And to customize output messages
  - Multi-way branches enable more categorization of data
- A branch can check that a file was successfully opened
  - Because a user may have mis-typed the name on input
- Loops are initially for repeating input, or output, or input-output

What Do We Currently Teach?

- Loops are initially for repeating input, or output, or input-output

A commonly taught looping pattern (priming read, sentinel)

```
cin >> month >> day;                        // Get a date--priming read
while ( !(month == 2 && day == 31) )
{
  :                                         // Process it
  cin >> month >> day;                      // Get the next date
}
```

What Do We Currently Teach?

- Arrays hold numbers (but we have to know how many ahead of time)
  - And sometimes other stuff
  - And play well with loops - especially count controlled loops
  - Including nested loops when there are multiple dimensions
- Structs hold other stuff
  - Which can include numbers
  - And they don’t work well with loops

What Do We Currently Teach?

- Objects are structs with functions attached, then it gets weird
  - Inheritance, polymorphism, generics, and overloading, oh my!
- Often taught in a bit of a rush, all at once, as something advanced
  - But not used generally
  - At least until we get to data structures

What Do We Currently Teach?

- Stacks are a cool use of objects!
  - And queues! And heaps! And priority queues! Sets! Maps! Hashing!
  - And trees! Search trees! AVL trees! Red-black trees!
- Searching is fun too! Linear, and binary, and hashed, tree traversals, and…
- Sorting is more fun! Bubble, insertion, selection, merge, quick, heap…
- Graphs are neat….
  - but mostly as a concept…

And all of it is based on sequential algorithms and complexity models
No parallelism, no asynchrony, no distribution, weak scalability

What Do We Currently Teach?

- Pattern: Build common collection types from scratch before using them
  - To see how they work
  - Usually cutting corners for robustness, efficiency, security
- Exemplifies weak API engineering
  - Teaches that testing is an afterthought, rather than a design goal
  - Ignores issues of security

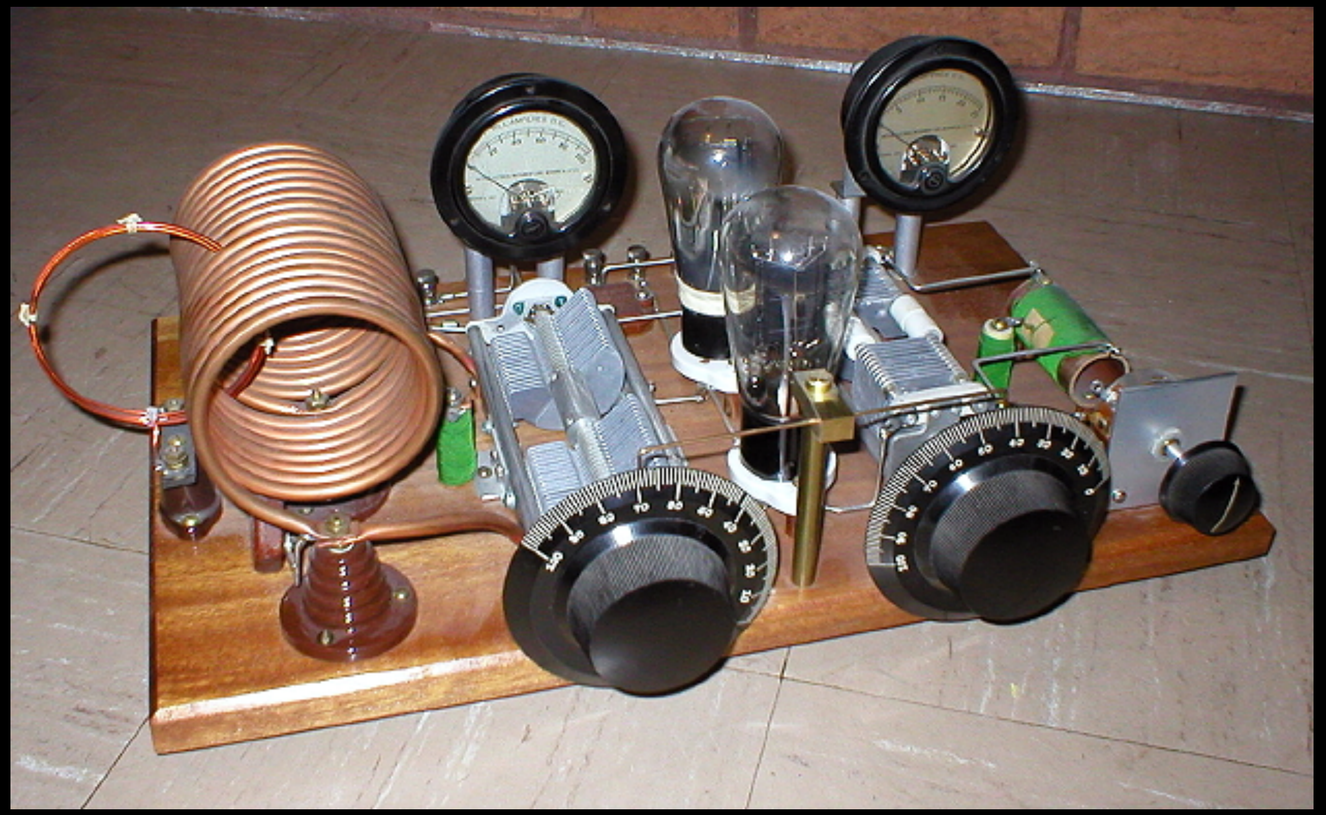
Creates a disposition that is anathema to modern practice

What Are We Teaching Students to Program?

What is the target of our intro to computational problem solving?

What would you say if…

EE majors still had to master designing circuits like these



Before they could learn about integrated circuits?

Perhaps…

- They’re wasting their student’s time
  - They’re wasting the early classes on irrelevant skills and design patterns
- They’re not preparing students for the real world
  - Their students won’t be able to function in careers in the industry
- They’re out of touch with reality
  - Students will leave the major because it isn’t about what they expected
- They need to move into the 21st century and look to the future
- Students are being harmed - this has to stop

What is the Modern Model?

What must we prepare our students to work with?

- APIs for everything
- Most modern languages even wrap raw types in classes
- Data we encounter every day includes
  - Video, image, sound, animations, graphics, icons
  - File paths, URLs, search results, web pages
  - Social media posts, news stories, calendar entries
  - Maps, GPS coordinates
- All with numerous features for controlling and interacting with them

But where do we find the time?

Common question from faculty

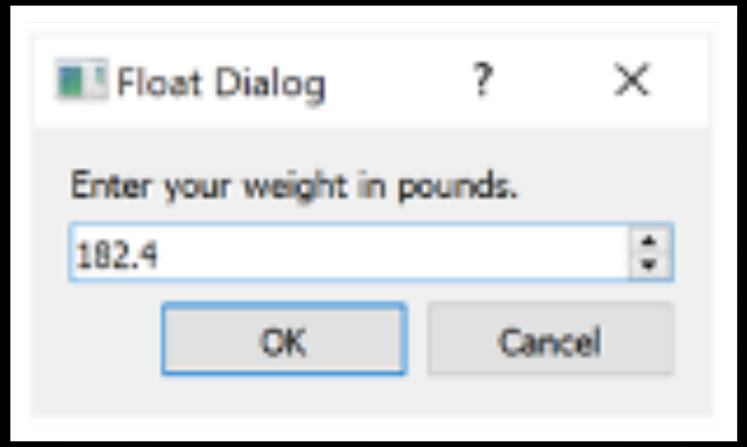
- The curriculum is already full
  - But it is full of hours of teaching obsolete patterns and paradigms
- Re-envisioning the target computing model for introductory computer science would identify a different set of goals
  - After removing the obsolete approaches, there is much room for new and engaging content and experiences

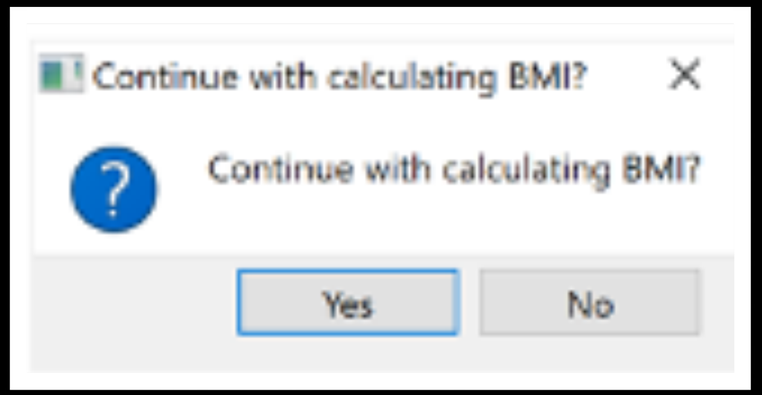
What is actually used in practice?

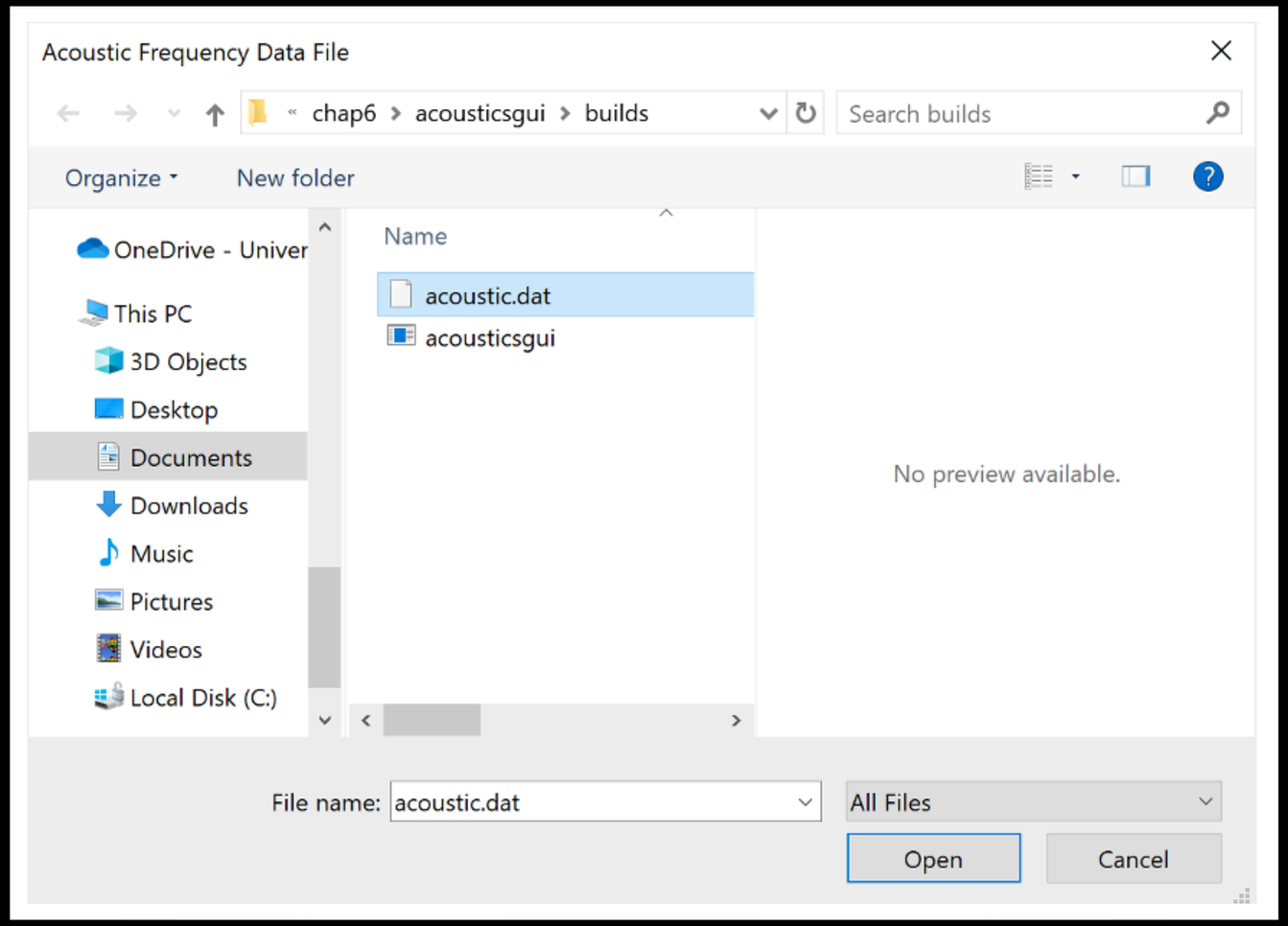
What must we prepare our students to work with?

- Output is in windows
  - A window is an object with an API that controls its appearance
  - Things can be put into windows in different arrangements
    - Arrangement also conveys information
  - Output can necessitate a chain of type conversions

With a Modern Model







Text-based data validation patterns can often be replaced by the choice of input object

With a Modern Model

- Repetitive input uses a window with event-generating objects
- An important concept is connecting an event handler to an event source
  - Event handling avoids many console-driven loop patterns
  - Event handling introduces the concept of asynchrony as fundamental
    - Breaks rigidity of sequential thinking - prepares ground for multithreading
  - Notice how early and naturally this PDC concept appears
    - Even before loop

With a Modern Model

- We have collection types (e.g., Vector, List, Dictionary, Hashmap, etc.)
- Key concept: Collection types support iterators
- Key concept: Languages have for-all loops that iterate over collections
- Note that “iterate” could be a parallel implementation
  - The first and most natural form of loop is parallelizable

```
List<Event> items = events.getItems();
String eventList = "Upcoming Events";
for (Event event : items)
  eventList += "\n" + event.getStart().getDateTime() + " " + event.getSummary();
JOptionPane.showMessageDialog(null, eventList);
```

With a Modern Model

- Collections and for-all loops subsume many of the traditional loop patterns
  - e.g., sub-array processing, testing for null pointers, enlarging space
  - Collections know their size and manage it dynamically
    - Seeing the value of doing it well motivates later implementation study
- The for-all pattern further breaks rigidity of sequential thinking
  - When ordering doesn’t matter in a collective for, it’s effectively parallel
- Students can build very powerful applications at this point

Summary

- Current curriculum was designed for an old computing model
- Builds from an abstraction that is close to early hardware
- Modern practice uses abstractions at a much higher level
- Gap is too great for a bottom-up approach that tries to anticipate every use, but can be bridged with a selective top-down approach
- Industry needs engineers who can work with APIs, work with PDC systems, design for test and security, sometimes build from scratch

Early PDC education isn’t an end in itself — it is something that arises naturally from changing the target computing model of CS education to something that represents modern systems

CS 1 Students Bring a Beginner's Mind

- No preconceived ideas of algorithms as sequential
  - Although HS AP courses may have already done some damage
- For them, computing is defined by familiar experiences
  - Multiple windows active at once, reformatting while typing, etc.
  - Phone can handle texting, calling, video simultaneously
  - Social media allows many users to access info at once
- They want to know how these common artifacts work

How Does a “Multicore” Work?

Answering with an Unplugged Activity

- List of numbers on board (e.g., 1 - 10) - represents a list in memory
- Two students designated as “cores” to do processing of list values
  - One core will add 1 to a value, the other will add 2
- Two more students designated as memory access “gophers”
- A gopher goes to the board, writes a number on a slip of paper
- Takes paper to core, who processes it, returns it to gopher
- Gopher returns to board, erases old value, writes new value in its place

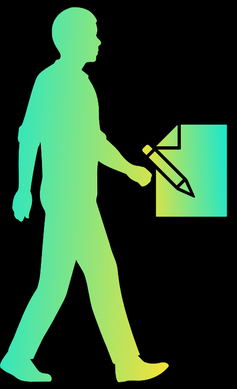
Memory
1
2
3
4
5
6
7
8
9
10

Setup

Cores

+1

+2



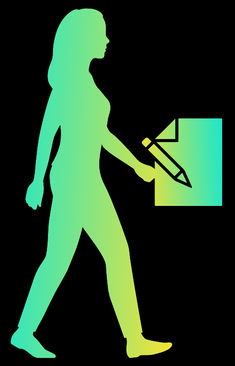
Gophers

Memory
1
2
3
4
5
6
7
8
9
10

Setup

Cores

+1



+2

Gophers

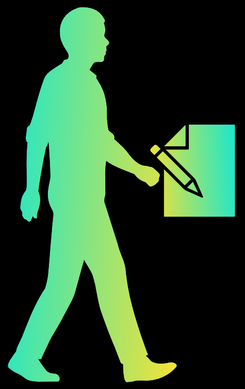
Memory
1
2
3
4
5
6
7
8
9
10

Setup

Cores

+1

+2



Gophers

Memory
2
2
3
4
5
6
7
8
9
10

Setup

Cores

+1

+2

Gophers

Memory
3
2
3
4
5
6
7
8
9
10

Try It

What Happens?

- Depending on speed of students
  - Values may be increased by 1, 2, or 3
  - The pattern may be consistent (e.g., if one student is consistently ahead)
  - Or it may be random
- Correct result would be that all numbers increased by 3

What Can We Do?

Brainstorm Ideas

What Can We Do?

- Students brainstorm ideas
  - Many will be equivalent to doing the operations serially
- Eventually (perhaps with some guidance) they may decide to split the list between the cores, then swap halves
- Then need to identify a coordination mechanism
  - A tag on each half of the list, indicating which core “owns” it
  - When a core finishes, it releases the tag, and the other can take it
- Try it — will often result in one gopher and core waiting in the middle

Implications

- We can do work faster with multiple cores
- Some speedup will be lost to coordination work (overhead)
- We could strategize work division in advance — not always possible
- Useful in real life — e.g., collaborative writing project, working on sections

How Does a User Interface Work?

- Modern interfaces are graphical
  - Command line may be comfortable for geeks, but not for beginners
- Modern CS1 can use GUI components as examples of OOD
- Frames, Buttons, Labels are great examples of objects
- e.g., Java ActionListeners introduce event-driven algorithm design
  - First step toward concurrency (non-sequential thinking)

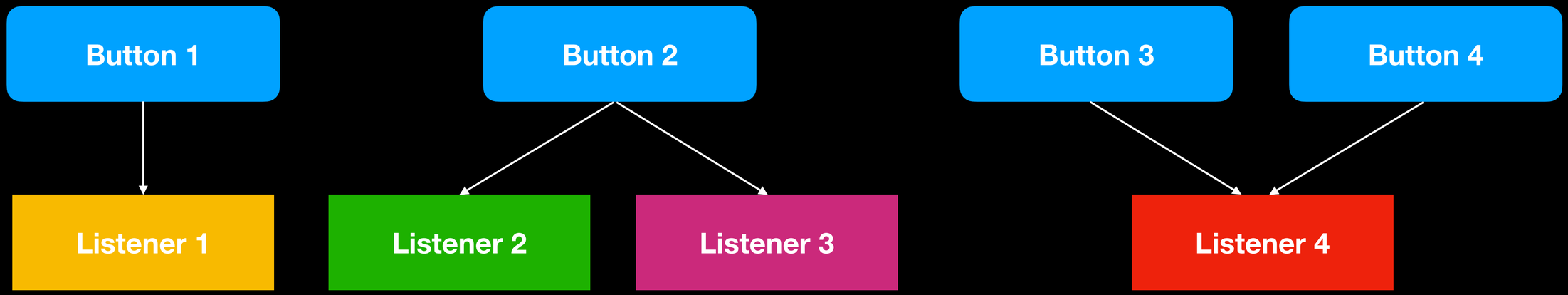
Java ActionListeners

- Java Swing components (simpler than FX, but could also use it)
- Active UI input components (e.g., buttons) can add (register) them
  - When the UI element is triggered, it notifies each registered listener
  - Although listeners are not concurrent, they can be imagined that way
  - Notification order for multiple listeners is not specified

Qt Signals and Slots

- Qt for C++, PyQt equivalent for Python
- Declare a button object with a title
- Define a function (do_it) to be called when button is clicked
- button.clicked.connect(do_it)
- As before, this is a familiar UI element, and the asynchronous handling of it makes sense - creating a mindset that this is a basic aspect of computational problem solving

ActionListeners



Clicking a button activates its listeners

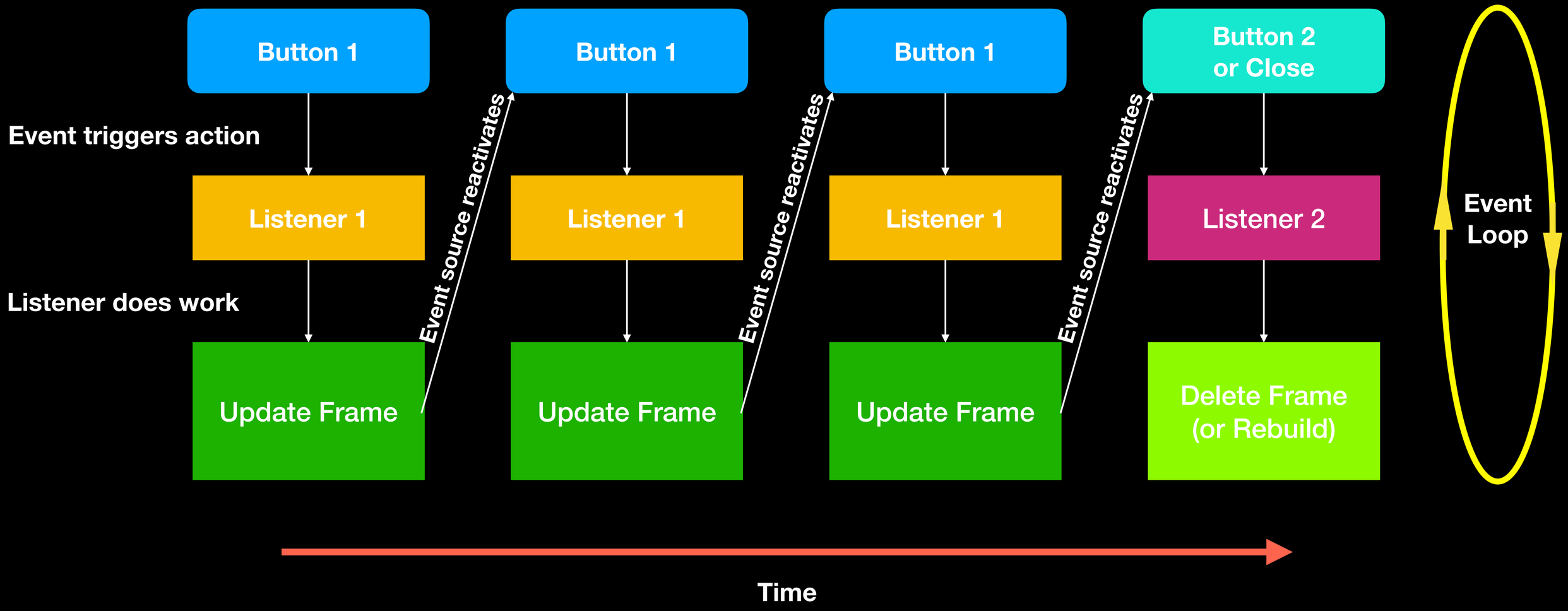
Button 1 executes 1 listener. Button 2 executes two listeners. Buttons 3 and 4 call the same listener at different times.

Side note: Listeners can identify the source of their activation, and respond differently

Event Loops

- With event-driven applications, some common loops disappear
  - while (notDone) {input, process, output, update notDone}
- Becomes a Frame with a close operation, and an ActionListener
- A button triggers processing of input, update of display
  - Either closing the window, or another button, triggers exit
- Learning event-driven patterns develops non-sequential algorithmic problem solving, and is good preparation for message passing parallelism

Event Loop


Button 1
Button 1
Button 1
Button 2
or Close
Event triggers action
Listener 1
Listener 1
Listener 1
Listener 2
Listener does work
Update Frame
Update Frame
Update Frame
Delete Frame
(or Rebuild)
Event source reactivates
Event source reactivates
Event source reactivates
Event
Loop
Time

Example (Two Buttons)

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class TwoThreadRace
{
    private static JFrame mainFrame;
    private static JLabel infoLabel;
    private static JLabel progressLabel;
    private static JButton leftButton;
    private static JButton rightButton;
    private static Integer progress = 0;

    public static void main(String[] args)
    {
        mainFrame = new JFrame("Two Thread Button Example");
        mainFrame.setSize(400, 400);
        mainFrame.setLayout(new GridLayout(2, 2));
        mainFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        infoLabel = new JLabel("Progress:", JLabel.RIGHT);
        progressLabel = new JLabel("0", JLabel.LEFT);
        leftButton = new JButton("+1");
        rightButton = new JButton("+2");
        mainFrame.add(infoLabel);
        mainFrame.add(progressLabel);
        mainFrame.add(leftButton);
        mainFrame.add(rightButton);
```

- Declare Frame, 2 Labels, 2 Buttons
- Set up Frame (title, size, layout — 2x2 grid, close action)
- Instantiate Labels and Buttons
- Add them to the Frame
  - Fill grid cells in reading order

TwoButtons Continued

```
        leftButton.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e) {
                int temp = progress;
                pause();
                progress = temp + 1;
                progressLabel.setText(" " + progress);
            }
        });

        rightButton.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e) {
                int temp = progress;
                pause();
                progress = temp + 2;
                progressLabel.setText(" " + progress);
            }
        });

        mainFrame.setVisible(true);
    }

    public static void pause(){
        try{Thread.sleep(1000);
        } catch (InterruptedException e) { }
    }
}
```

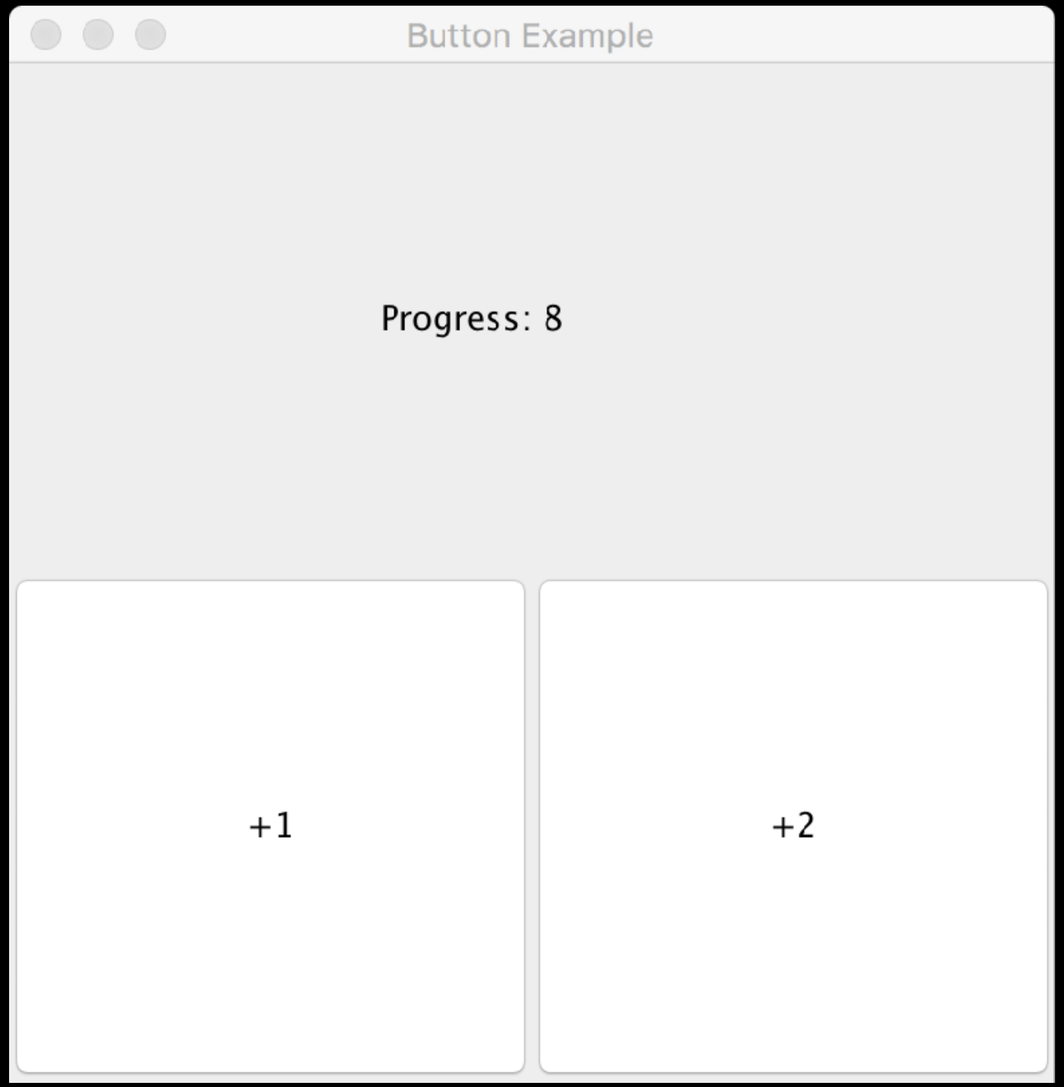
- Create a listener for each button
- Here we simulate doing time consuming work using Pause
- We also split the access to the global variable and its update across the work
- Make the Frame visible
- Pause calls Thread.sleep for 1 sec

Sample Run of TwoButton

- What happens when we rapidly click different buttons?

Effects

- No immediate update
  - Long delay for multiple clicks
- Some updates are skipped
- Update of Progress value happens in proper order, and correctly
  - Even though we allowed plenty of time for listeners to get and update value incorrectly



Why?

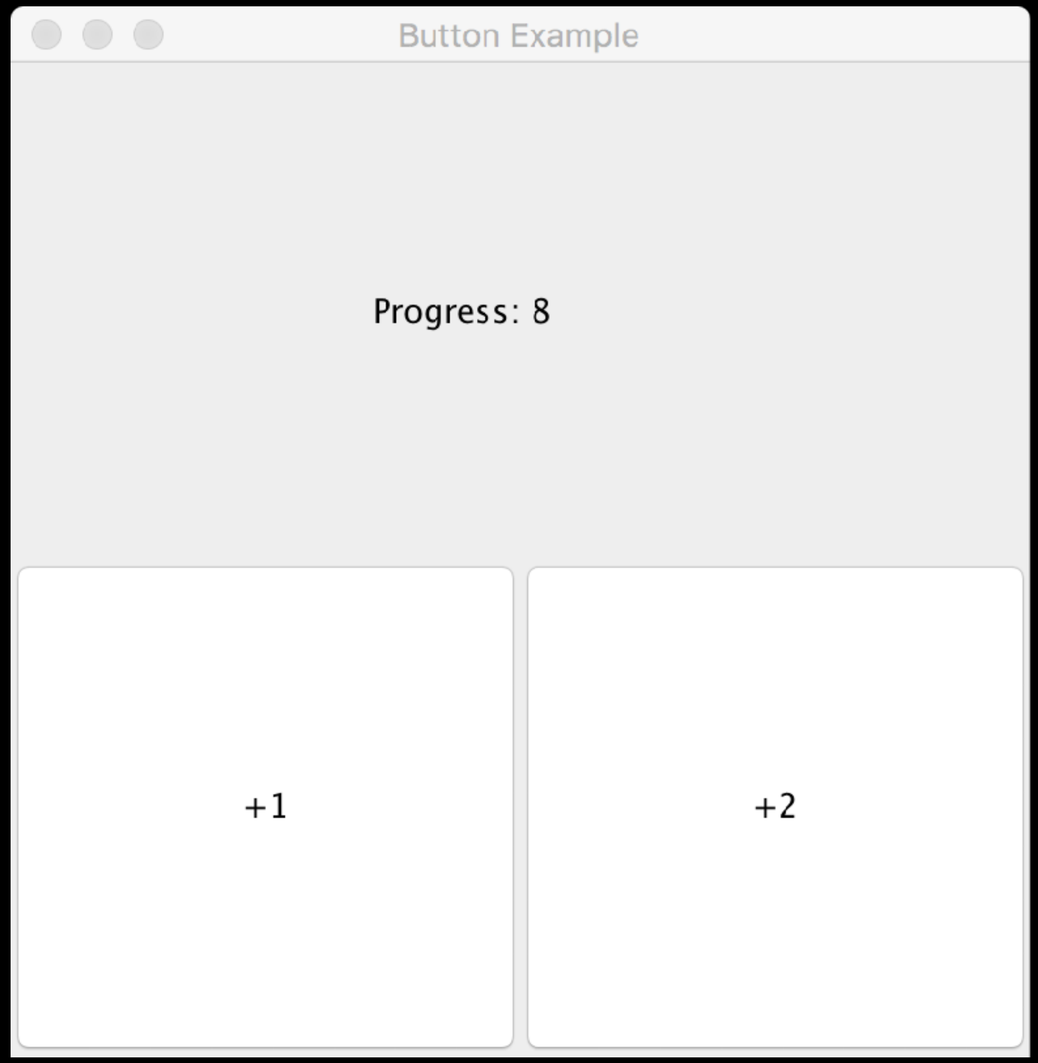
- There is a single event dispatch thread
- Each event action is placed on a queue, then executed in order
  - That’s why we can click repeatedly before an update
  - Screen updates happen when the queue empties
- Although there are multiple action listeners, they don’t work in parallel
  - That’s why the time adds up
- But we can also see evidence of multiple threads executing concurrently…

Where’s the Concurrency?

- There is the event dispatch thread (we already know this)
  - It calls the ActionListeners in order, to execute sequentially
- But there is also an event enqueuing thread
  - Something has to watch for clicks, associate them with UI elements, and put the corresponding ActionListener event on the queue
  - That has to happen concurrently with the work of the event dispatch thread

What’s the Problem?

- No immediate update
  - Long delay for multiple clicks
- Some updates are skipped
- In the meantime, the display can seem frozen for the user



What's the Solution?

- For each action type, define a thread class with the code to do the work
- Instantiate a thread to do the work of each ActionListener
  - The ActionListener just instantiates a worker and executes it
- When the event dispatch thread runs the ActionListener, it quickly spawns the new thread and finishes
  - The separate thread then does the work concurrently with further event dispatching

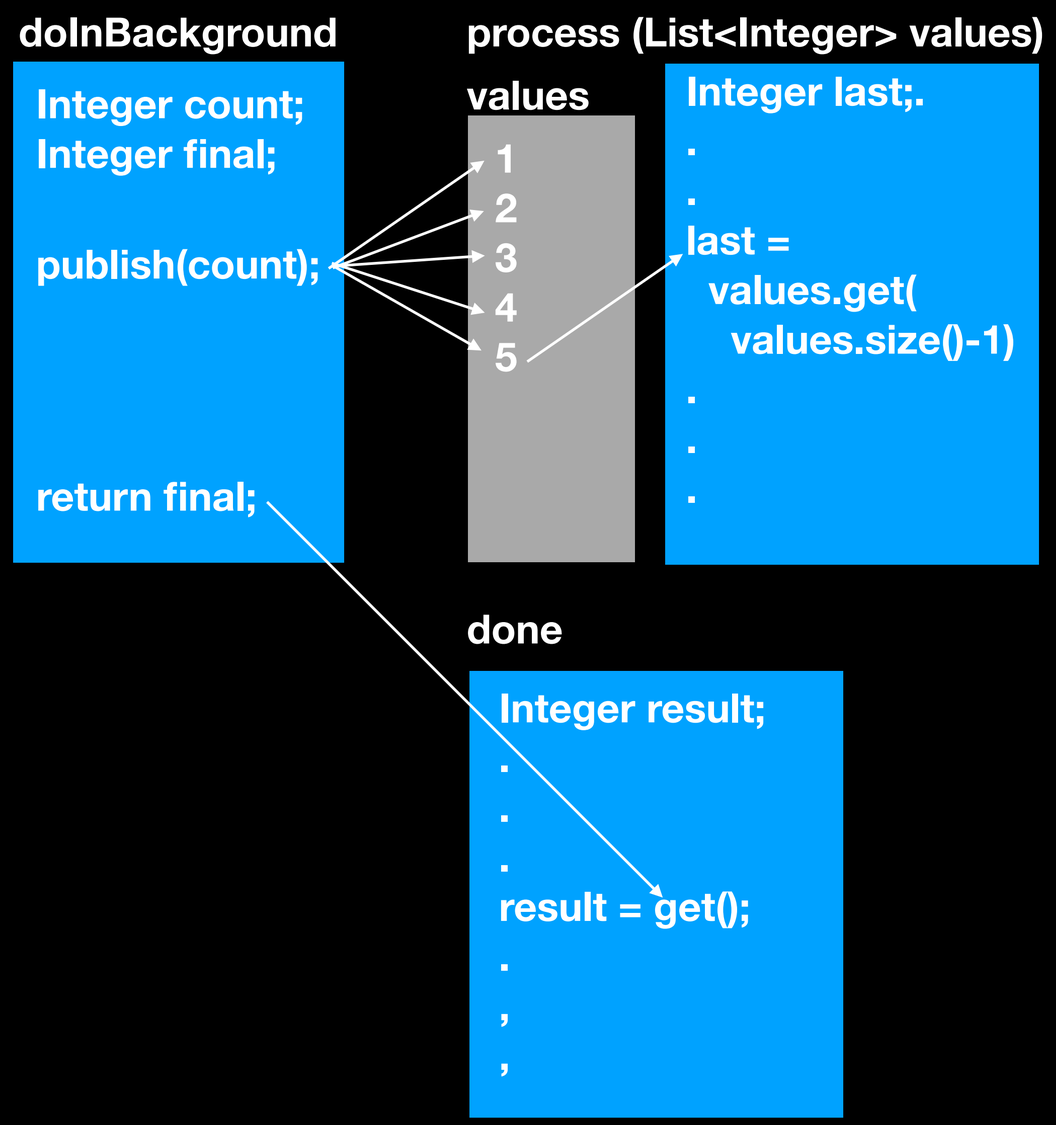
SwingWorker

- Abstract class with two template parameters
- Hides complexity of Java Thread class — specialized to Swing
- Provide methods that we override: doInBackground, done, process
  - doInBackground does work, publishes results, returns value on exit
- First template parameter is type of value returned on exit
- Second template parameter is type of values to be published

Where do returned and published values go?

Value Recipients

- When doInBackground calls publish, values go on the list declared as a parameter to process
- process asynchronously retrieves the values from the list, working concurrently with doInBackground
- The return value from doInBackground is available to done via a call to get()



Building Workers

```
class MainWorker1 extends SwingWorker<Integer, Integer>
{
   protected Integer doInBackground() throws Exception, InterruptedException
   {
      // Do a time-consuming task.
      Thread.sleep(1000);
      progress = progress + 1;
      return progress;
   }

   protected void done()
   {
      try
      { progressLabel.setText(" " + get());
      } catch (Exception e){ }
   }
}

class MainWorker2 extends SwingWorker<Integer, Integer>
{
   protected Integer doInBackground() throws Exception, InterruptedException
   {
      // Do a time-consuming task.
      Thread.sleep(1000);
      progress = progress + 2;
      return progress;
   }

   protected void done()
   {
      try
      { progressLabel.setText(" " + get());
      } catch (Exception e){ }
   }
}
```

- Same code for declarations, etc.
- Then define two workers
  - Each has two methods
- doInBackground is where work (sleep) happens
- done handles wrap-up when doInBackground exits
- process isn’t used here, but could handle a series of values and UI updates
- In this example, we want to illustrate threads handling individual work units and exiting

Instantiate and Execute

```
    leftButton.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e) {
            new MainWorker1().execute();
        }
    });

    rightButton.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e) {
            new MainWorker2().execute();
        }
    });

    mainFrame.setVisible(true);
  }
}
```

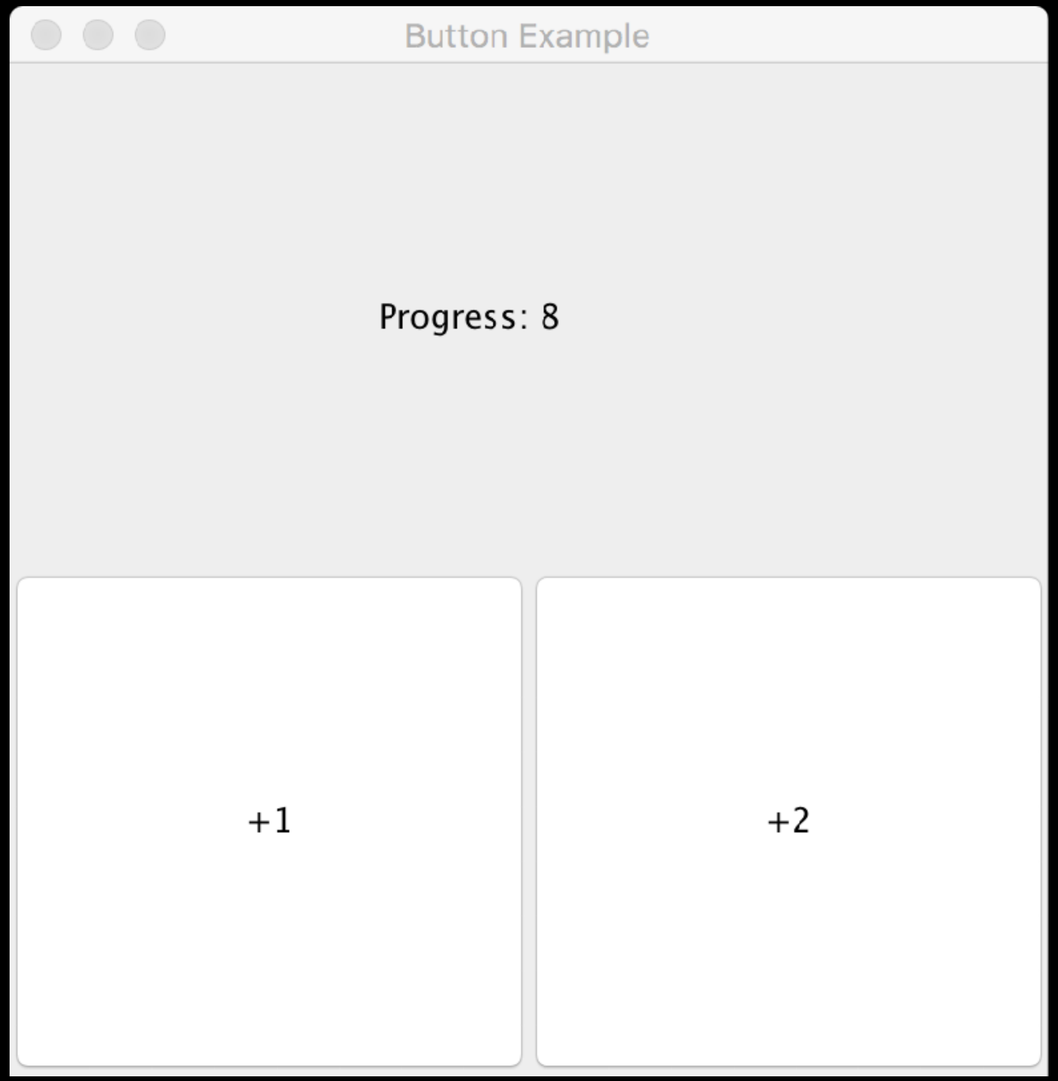
- ActionListener now does one thing:
  - Instantiate a worker and invoke its execute method
- Event dispatch can do this quickly
- Each button click launches a thread of the corresponding type to do its work
- Note that doInBackground reads and updates progress in one statement

Sample Run of TwoThread

- What happens when we rapidly click different buttons?

Effects

- The value updates while buttons are being clicked
- It increments in the proper order
- The total is correct
- Even though the “work” adds up to many seconds, it completes in less time
- Obviously exploiting concurrency



Problem Solved?

- Not so fast…
- We are updating a global variable
- The time to read, update, and save the value is tiny in comparison to the work time
  - But there is still a small chance that two threads could read the same value, update it, and overwrite it
  - To see this in action, we increase the odds

Two Buttons with Access Race

```
class MainWorker1 extends SwingWorker<Integer, Integer>
{
   protected Integer doInBackground() throws Exception, InterruptedException
   {
      Integer temp = progress;
       // Do a time-consuming task.
      Thread.sleep(1000);
      progress = temp + 1;
      return progress;
   }

   protected void done()
   {
      try
      { progressLabel.setText(" " + get());
      } catch (Exception e){ }
   }
}

class MainWorker2 extends SwingWorker<Integer, Integer>
{
   protected Integer doInBackground() throws Exception, InterruptedException
   {
      Integer temp = progress;
      // Do a time-consuming task.
      Thread.sleep(1000);
      progress = temp + 2;
      return progress;
   }

   protected void done()
   {
      try
      { progressLabel.setText(" " + get());
      } catch (Exception e){ }
   }
}
```

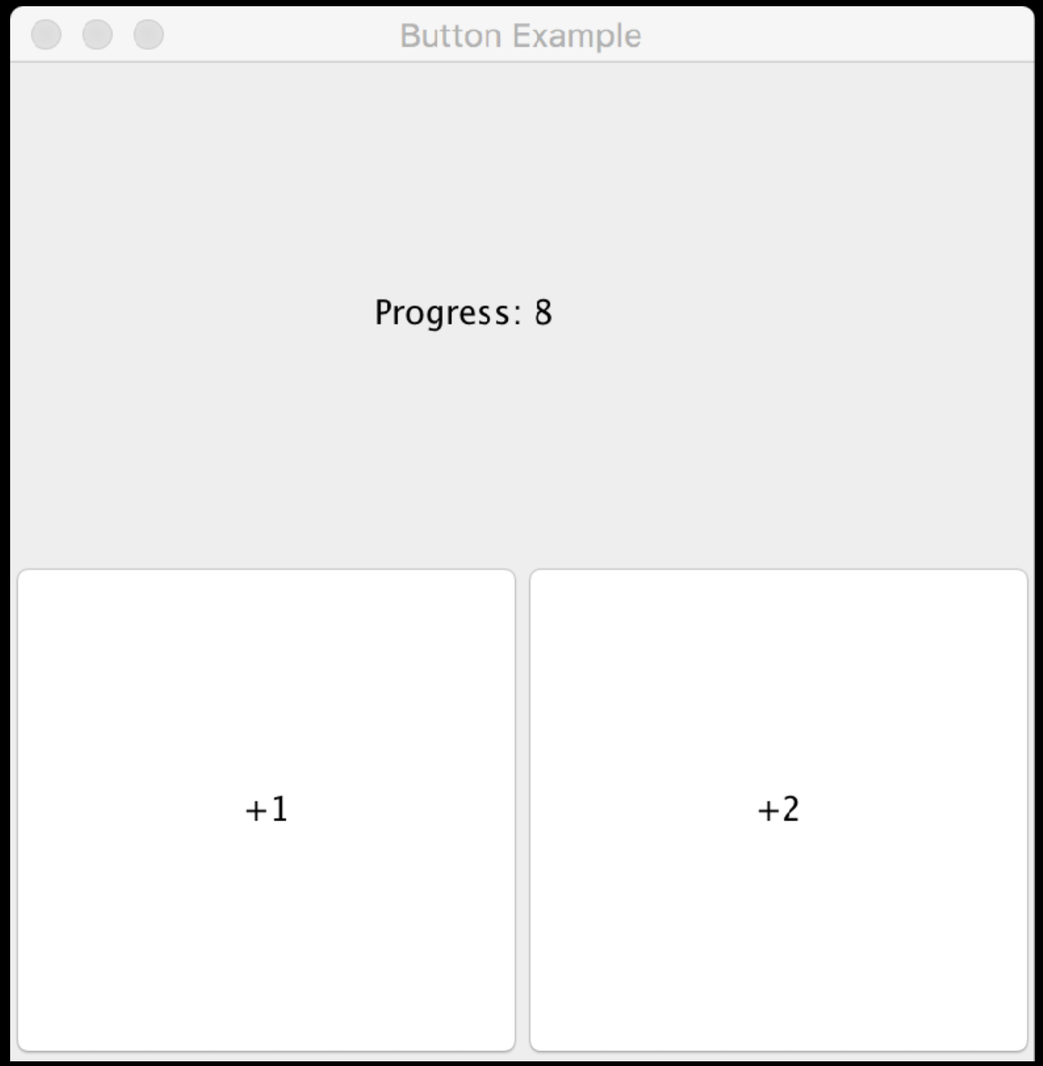
- Only small change to each worker
- Read progress before work
- Update progress after work
- We did this in the original version, but the event queue saved us

Sample Run of TwoThreadRace

- What happens when we rapidly click different buttons?

Effects

- The value updates while buttons are being clicked
- It increments in arbitrary order
- Sometimes it seems to decrement
  - Actually write-after-write hazard
- The total is incorrect



How Do We Avoid It?

- We could go deeper into the Java Thread class and use a synchronized method to update progress
- Fortunately, SwingWorker makes it easier:
  - done always executes in the GUI event thread (so does process)
  - Multiple executions of done thus happen sequentially
- If we put the update in done, then it will avoid hazards
- The workers still do the heavy work, and can return the update increment

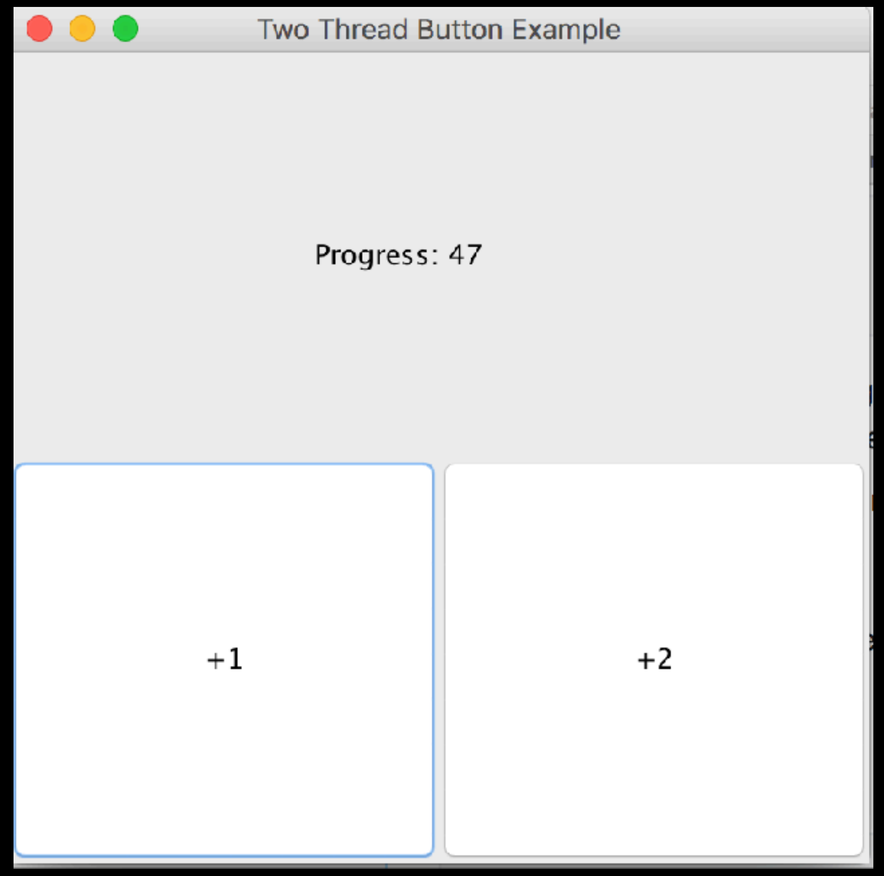
Building Workers

```
class MainWorker1 extends SwingWorker<Integer, Integer>
{
  protected Integer doInBackground() throws Exception, InterruptedException
  {
      // Do a time-consuming task.
     Thread.sleep(1000);
     return 1;
  }

  protected void done()
  {
     progress = progress + get();
     try
     { progressLabel.setText(" " + progress);
     } catch (Exception e){ }
  }
}

class MainWorker2 extends SwingWorker<Integer, Integer>
{
  protected Integer doInBackground() throws Exception, InterruptedException
  {
     // Do a time-consuming task.
     Thread.sleep(1000);
     return 2;
  }

  protected void done()
  {
   progress = progress + get();
     try
     { progressLabel.setText(" " + progress);
     } catch (Exception e){ }
  }
}
```

- Each doInBackground does the work (sleep) then returns the increment
- Each done increments progress and updates the display
- The work is done concurrently
- Display update still happens quickly
- If work was much longer, we could publish the increment and process it

Sample Run of TwoThreadNoRace

- The value updates while buttons are being clicked
- It increments in order
- The total is correct
- The overall time is less than the total work time



General Rule

- Use concurrency for large units of independent work
- Minimize access to shared variables
- Serialize shared access with as little overhead as possible
  - Note that updating the display was also a shared access all along

Questions?

Break!