What's an event?
Things happen

And when they happen, you want to know about it

User interface programming
Button/widget clicks, mouse motion, keyboard input, window resize, ...

Network programming
Message arrival, connection success/failure, network status change, ...

And more
Clock “ticks”, battery/charging status, ...

Common idea: when thing might happen, please call me!
Many terms for this sort of thing

Callbacks, Observers, Listeners, EventHandlers, ...

Same basic idea: you register some code to run when something happens
This stuff is used everywhere

java.util

Interface EventListener

• All Known Subinterfaces:

All Known Implementing Classes:
AbstractAction, AbstractButton.ButtonChangeListener, AWTEVENTLISTENERProxy, AWTEVENTMulticaster, BasicButtonListener, BasicColorChooserUI.PropertyHandler, BasicComboBoxEditor,BasicComboBoxEditor.URIResource, BasicComboBoxUI.FocusHandler, BasicComboBoxUI.ItemHandler,BasicComboBoxUI.KeyHandler, BasicComboBoxUI.ListDataHandler, BasicComboBoxUI.PropertyChangeHandler,BasicComboBoxPopup.InvocationKeyHandler, BasicComboBoxPopup.InvocationMouseHandler,BasicComboBoxPopup.ItemHandler, BasicComboBoxPopup.ListDataHandler, BasicComboBoxPopup.ListMouseHandler,BasicComboBoxPopup.ListMouseMotionHandler,BasicComboBoxPopup.ListSelectionHandler, BasicComboBoxPopup.PropertyChangeHandler, BasicDesktopIconUI.MouseInputHandler,BasicDesktopPaneUI.CloseAction, BasicDesktopPaneUI.MaximizeAction, BasicDesktopPaneUI.MinimizeAction, BasicDesktopPaneUI.NavigateAction, BasicDesktopPaneUI.OpenAction, BasicDirectoryModel,BasicFileChooserUI.ApproveSelectionAction, BasicFileChooserUI.CancelSelectionAction, BasicFileChooserUI.ChangeToParentDirectoryAction, BasicFileChooserUI.DoubleClickListener,BasicFileChooserUI.GoHomeAction, BasicFileChooserUI.NewFolderAction,BasicFileChooserUI.SelectionListener, BasicFileChooserUI.UpdateAction, BasicInternalFrameTitlePane.CloseAction,BasicInternalFrameTitlePane.IconifyAction,
Typical use (Android)

“Find” your UI widgets, register listeners on them

```java
liteButton = (RadioButton) findViewById(R.id.liteButton);
toolButton = (RadioButton) findViewById(R.id.toolButton);
numbersButton = (RadioButton) findViewById(R.id.numbersButton);
clockView = (MyViewAnim) findViewById(R.id.surfaceView);
secondsSwitch = (Switch) findViewById(R.id.showSeconds);
dayDateSwitch = (Switch) findViewById(R.id.showDayDate);

View.OnClickListener myListener = new View.OnClickListener() {
    public void onClick(View v) {
        // fetch state from UI widgets, redraw screen
    }
};

liteButton.setOnClickListener(myListener);
toolButton.setOnClickListener(myListener);
numbersButton.setOnClickListener(myListener);
secondsSwitch.setOnClickListener(myListener);
dayDateSwitch.setOnClickListener(myListener);
```
Typical use (Android)

“Find” your UI widgets, register listeners on them

```java
liteButton = (RadioButton) findViewById(R.id.liteButton);
toolButton = (RadioButton) findViewById(R.id.toolButton);
numbersButton = (RadioButton) findViewById(R.id.numbersButton);
clockView = (MyViewAnim) findViewById(R.id.surfaceView);
secondsSwitch = (Switch) findViewById(R.id.showSeconds);
dayDateSwitch = (Switch) findViewById(R.id.showDayDate);

View.OnClickListener myListener =
    new View.OnClickListener() {
    public void onClick(View v) {
        // fetch state from UI widgets, redraw screen
    }
};

liteButton.setOnClickListener(myListener);
toolButton.setOnClickListener(myListener);
numbersButton.setOnClickListener(myListener);
secondsSwitch.setOnClickListener(myListener);
dayDateSwitch.setOnClickListener(myListener);
```

Find the widget, set its listener.
Typical use (Android)

“Find” your UI widgets, register listeners on them

```java
liteButton = (RadioButton) findViewById(R.id.liteButton);
toolButton = (RadioButton) findViewById(R.id.toolButton);
numbersButton = (RadioButton) findViewById(R.id.numbersButton);
clockView = (MyViewAnim) findViewById(R.id.surfaceView);
secondsSwitch = (Switch) findViewById(R.id.showSeconds);
dayDateSwitch = (Switch) findViewById(R.id.showDayDate);

View.OnClickListener myListener = new View.OnClickListener() {
    public void onClick(View v) {
        // fetch state from UI widgets, redraw screen
    }
};

liteButton.setOnClickListener(myListener);
toolButton.setOnClickListener(myListener);
numbersButton.setOnClickListener(myListener);
secondsSwitch.setOnClickListener(myListener);
dayDateSwitch.setOnClickListener(myListener);
```

Shared “listener” (anon. inner class, Java7’s almost-lambda)
When we see the URL "/hello" or "/hello2", run the given lambda:

```java
public class SimpleServer {
    private static final String TAG = "SimpleServer";

    public static void main(String[] args) {
        get("/hello", (request, response) -> "Hello, world");

        get("/hello2", (request, response) ->
            html().with(
                head().with(title("Hello, world")),
                body().with(
                    h1("Hello, world"),
                    p("This is some introductory test"))));
    }
}
```
Closer to home: The Spark web server

When we see the URL "/hello" or "/hello2", run the given lambda:

```java
public class SimpleServer {
    private static final String TAG = "SimpleServer";

    public static void main(String[] args) {
        get("/hello", (request, response) -> "Hello, world");

        get("/hello2", (request, response) ->
            html().with(
                head().with(title("Hello, world")),
                body().with(
                    h1("Hello, world"),
                    p("This is some introductory test")));
    }
}
```
In a world with mutation...

Events are commonly used to respond to changes in state

Scoreboard on top
Listen on changes to player “health”
Listen to clock, display remaining time

Players
Listen to status of other player
(e.g., “if they’re suffering damage, laugh!”)
Why listeners and mutation are awful

Dependencies kill you
You listen on one class, and when it changes you update yourself
Somebody else listens on you and noticed that you changed, so they update
What if the first class was listening on the third?

It’s too easy to get weird circular dependencies
It’s also really confusing to understand the order in which things occur
Something you didn’t expect can trigger a cascade of listeners being called
But what about a functional world?

There are several general-purpose “functional-reactive programming” (FRP) frameworks like RxJava
One of the better introductions is here: 
https://www.bignerdranch.com/blog/what-is-functional-reactive-programming/

We’re not going to go there in Comp215
These frameworks generally deal with concurrent / multithreaded computation
(That’s for Comp322 and/or other classes)

If you do see “functional-reactive” code, it looks a lot like lazy list code
Filters, maps, etc. on events that haven’t happened yet.
“Subscribe” to events and emit more events.
You’re already all set to (mostly) understand functional-reactive code.
What about your adventure game?

How might you deal with locked doors?

Keys are functions that update doors from “locked” to “unlocked”
The key (or door) might verify they match properly

Door opening? Another function.
Check if the door is unlocked, etc.

Navigating through a door? Another function.
Check if there’s a door, if so, run its function.
Composes with the function to traverse any exit.
What about your adventure game?

How might you deal with the “pansy of perplexity”?

**Time-based effects**
World-state: tracks when the effect starts and ends, updates an “active” or “inactive” field.

The “pansy” function is the identity function unless it’s “activated”.

**Command-based effects**
The “pansy” transforms motion commands to different motion commands, or it’s the identity function.

Many different ways to accomplish the same goal!
How do you connect typed commands?

> HIT TROLL WITH SWORD

> HIT TROLL WITH FLOWER

One of these will be more effective than the other!

Special attributes on weapons that normal objects don’t have?

> HIT <target> WITH <item>

You need a general-purpose way to map from names to items

(Maybe an IMap<String, WorldItem> or whatever you call it.)
Expect to build a bunch of “registries”

```java
private static IMap<TokenType, CalcOp> registry =
    TreeMap.of(
        new KeyValue<>(TokenType.PLUS, RPNCalculator::add),
        new KeyValue<>(TokenType.TIMES, RPNCalculator::multiply),
        new KeyValue<>(TokenType.MINUS, RPNCalculator::subtract),
        new KeyValue<>(TokenType.DIVIDE, RPNCalculator::divide),
        new KeyValue<>(TokenType.DUP, RPNCalculator::dup),
        new KeyValue<>(TokenType.DROP, RPNCalculator::drop),
        new KeyValue<>(TokenType.SWAP, RPNCalculator::swap),
        new KeyValue<>(TokenType.EQUALS, RPNCalculator::noop),
        new KeyValue<>(TokenType.FAIL, RPNCalculator::fail),
        new KeyValue<>(TokenType.CLEAR, RPNCalculator::clear)
    );
```

Incredibly useful to find the functions that do what you need to do
You can update a registry as well

```java
private static IMap<TokenType, CalcOp> registry =
    TreeMap.of(
        new KeyValue<>(TokenType.PLUS, RPNCalculator::add),
        new KeyValue<>(TokenType.TIMES, RPNCalculator::multiply),
        new KeyValue<>(TokenType.MINUS, RPNCalculator::subtract),
        new KeyValue<>(TokenType.DIVIDE, RPNCalculator::divide),
        new KeyValue<>(TokenType.DUP, RPNCalculator::dup),
        new KeyValue<>(TokenType.DROP, RPNCalculator::drop),
        new KeyValue<>(TokenType.SWAP, RPNCalculator::swap),
        new KeyValue<>(TokenType.EQUALS, RPNCalculator::noop),
        new KeyValue<>(TokenType.FAIL, RPNCalculator::fail),
        new KeyValue<>(TokenType.CLEAR, RPNCalculator::clear)
    );
```

registry.update(...) gives you a new registry, leaves the original alone
Wrap existing functions, replace functions (temporarily?), etc.
Function composition is your friend

While you’re building your world (and reading your JSON input):
Function<WorldState,WorldState> evolver = myFunction.compose(oldEvolver);

Compose (or .andThen) your functions together and each does its thing

Support autonomous actors in the game (e.g., a thief or a monster)

Support state-changes that have broad effects (e.g., a light switch in one room that illuminates many rooms)
Don’t be afraid to stretch a bit

Maybe you want your functions to take two arguments
BiFunction<WorldState, UserCommand, WorldState>
(i.e., a function from the state of the world and the command to a new world state)
Or, maybe you want the current command to be just another field in the WorldState

Maybe you want your functions to use Optionals everywhere
Easier to deal with errors, as we did with the RPNCalculator

Maybe you want your functions to return Pairs of things
A new world state, plus whatever message the user will ultimately see.
Or maybe the message to the user is just a value in the WorldState.
(Making it trivial to hang onto previous messages, previous commands, etc.)
Testing, testing, testing!

Create very simple worlds for your tests
One room, one object in the room, etc.

Create very simple functions that operate on those rooms
Pick up an object, drop an object, etc.

Unit test them individually

Don’t compose them until they work on their own
But... but... performance!

A bunch of functions that “do nothing” chained together could be a problem
But only if there were a whole lot of them and they were running all the time

Your game only executes codes when the user types a command
If your response time is less than a second, you’re doing just fine
   And your computer can execute well over a billion CPU instructions per second!

Basically, don’t worry about it
When should you care then?

1) If you’re trying to redraw the screen at 60+Hz
   “Jank is any stuttering, juddering or just plain halting that users see when a site or app isn't keeping up with the refresh rate. Jank is the result of frames taking too long for a browser to make, and it negatively impacts your users and how they experience your site or app.”

   We’ll talk more about Android and dealing with jank on Nov 23 (Monday before Thanksgiving).

2) If you’re running a huge computation
   Think: Google-scale, where a performance boost means serious savings on the electricity bills.

3) If you’re running on a tiny computer
   Think: A pacemaker, where a performance boost means fewer surgeries to replace a dead battery.
Remaining time: Q&A on the project