Leibniz Universität Hannover

Mensch-Computer-Interaktion 2

Event Handling

Human-Computer Interaction Group

Prof. Dr. Michael Rohs
michael.rohs@hci.uni-hannover.de

Lectures

Session	Date	Topic	
1	6.4.	Introduction	
2	13.4.	Interaction elements	GUI toolkits, interaction techniques
3	20.4.	Event handling	
4	27.4.	Scene graphs	
5	4.5.	Interaction techniques	
	11.5.	no class (CHI)	
	18.5.	no class (spring break)	
6	25.5.	Experiments	design and analysis of experiments
7	1.6.	Data Analysis	
8	8.6.	Data Analysis	
9	15.6.	Visualization	
10	22.6.	Visualization	current topics beyond-desktop UIs
11	29.6.	Modeling interaction	
12	6.7.	Computer vision for interaction	
13	13.7.	Computer vision for interaction	

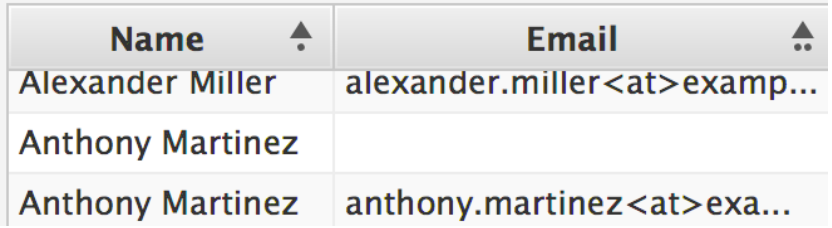
Klausur:
28.7.2016
8-11 Uhr
HG E214

Leibniz Universität Hannover

COMPLEX WIDGETS, EVENT HANDLING

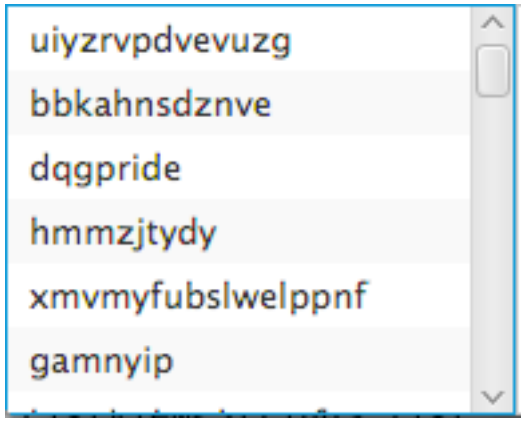
Collections of Items

- Interacting with collections of items
  - List view
  - Table view
  - Tree view
  - Tree table view



- ListView as a detailed example
  - The other widgets (controls) use the same general patterns

List View



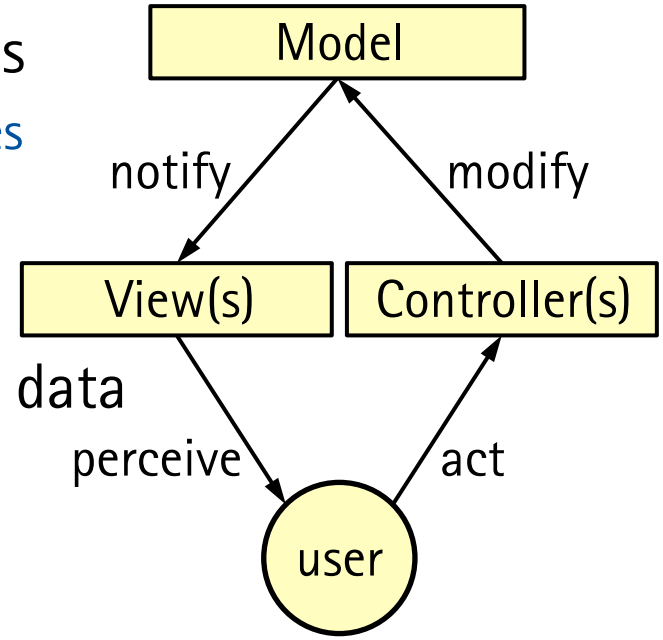
- Scrollable list of (editable) items
  - Vertical column or horizontal row
- Example

```
ListView<String> list = new ListView<>();
ObservableList<String> items = generateItems(100); // the list's data model
list.setItems(items);
list.setOrientation(Orientation.VERTICAL)
```

- ListView<T> is a generic type, where T is the type of the items
  - e.g., String or CheckBox
- The list view automatically observes changes in its data model
  - Observable data structures allow registering listeners

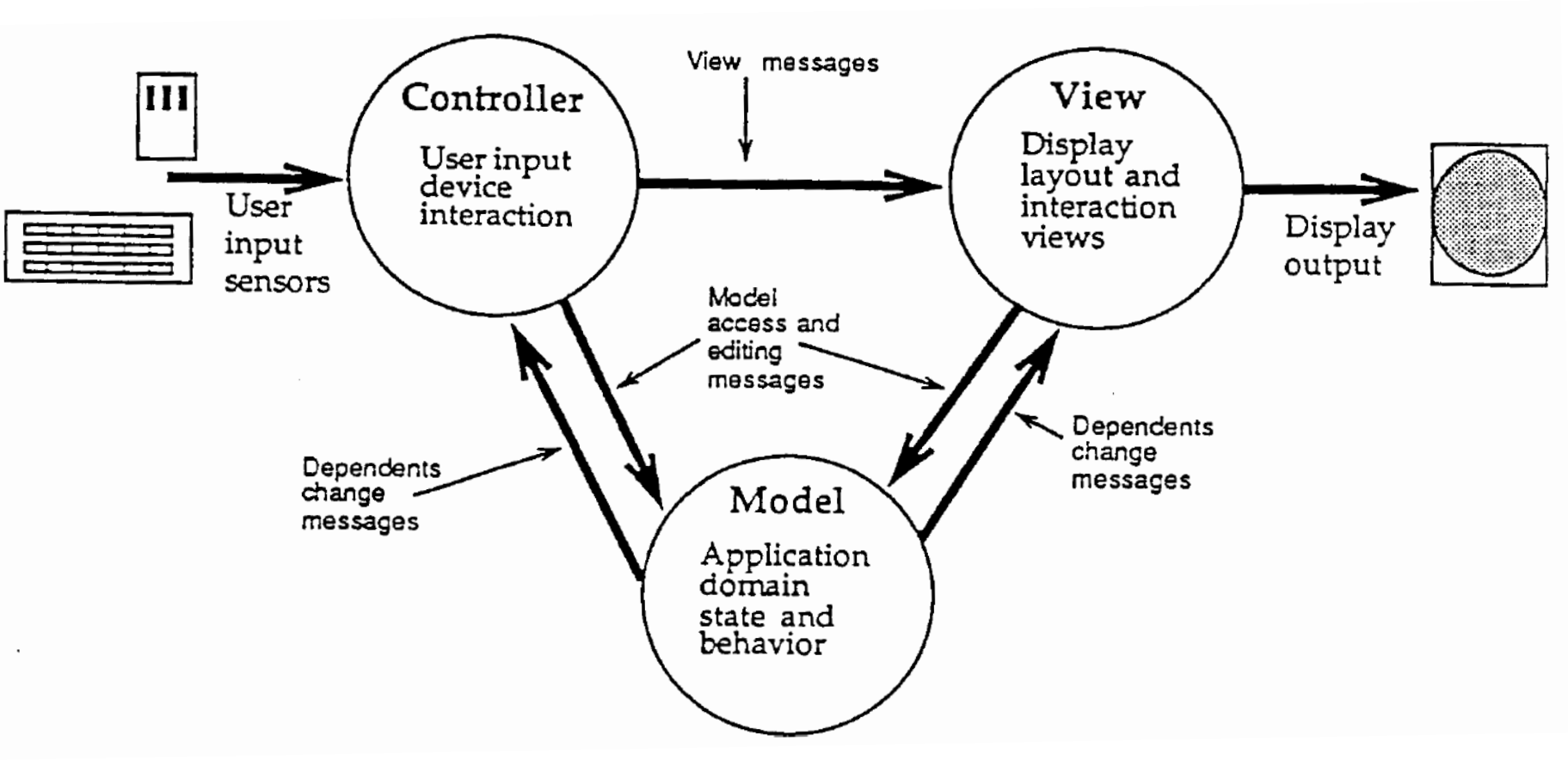
Model-View-Controller (MVC) Pattern

- Model: Domain-specific data & operations
  - Notifies views and controllers of state changes
- View(s): Output to show the model's data
  - There may be different views on the data
- Controller(s): Input to modify the model's data
  - There may be different controllers
  - The controller may also control the view
- Characteristics
  - Often, view and controller merge (as in ListView)
  - Flexibility, modularity, separation of concerns
  - Widely used in implementing user interfaces



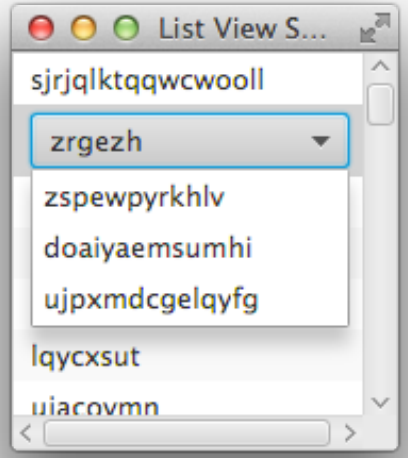
Krasner, Pope. A cookbook for using the model–view controller user interface paradigm in Smalltalk-80. The Journal of Obj. Tech., Aug–Sep 1988.

Model-View-Controller (MVC) Pattern



Krasner, Pope. A cookbook for using the model–view controller user interface paradigm in Smalltalk-80. The Journal of Obj. Tech., Aug–Sep 1988.

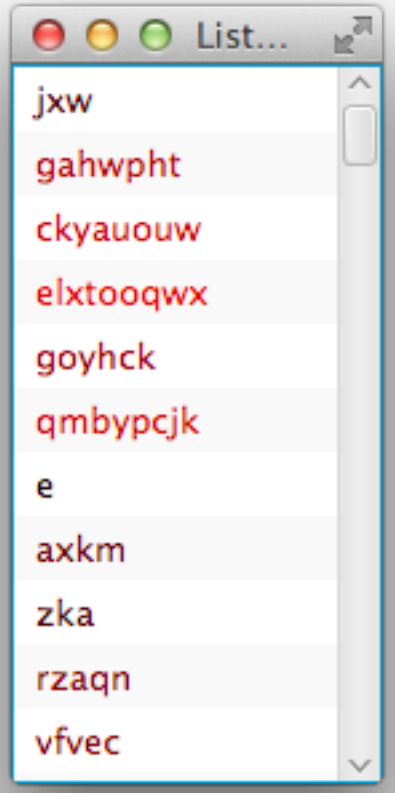
List View Cells



- Each row is rendered using a "cell"
  - Base class: ListCell
  - Further classes: CheckBoxListCell, ComboBoxListCell, TextFieldListCell, etc.
- List view uses a "cell factory" to generate new cells
  - List model may be very large
  - Only as many cells generated as visible on the screen
  - Cells are reused when scrolling
- Providing a specific cell class (and a factory for producing cells)
  - Example: ComboBoxListCell (rendered as label, when not editing)

```
listView.setEditable(true);
listView.setCellFactory(ComboBoxListCell.forListView(generateItems(3)));
```

List View Cells



- Customized cell rendering
  - Create subclass of ListCell<T>
  - Set the cell factory for producing cells
- Example: Change cell text color

```
private static class ColoredCell extends ListCell<String> {
    public void updateItem(String item, boolean empty) {
        super.updateItem(item, empty);
        if (item == null || empty) { setText(""); }
        else {
            double red = 0.1 + Math.min(0.9, (item.length() - 1) / 10.0);
            setTextFill(new Color(red, 0, 0, 1));
            setText(item); }
    }  }
```

```
listView.setCellFactory(l -> new ColoredCell());
```

List View Focus and Selection

- Interacting with the ListView changes its state

- Focus model represents the focused item
  - Get focused item, move focus, focus specific item

```
FocusModel<String> focusModel = listView.getFocusModel();
```

- Selection model represents the selected item(s)
  - Get selected item(s), select specific item(s)

```
MultipleSelectionModel<String> msm = listView.getSelectionModel();
```

  - By default, selection of a single item only, to select multiple items:

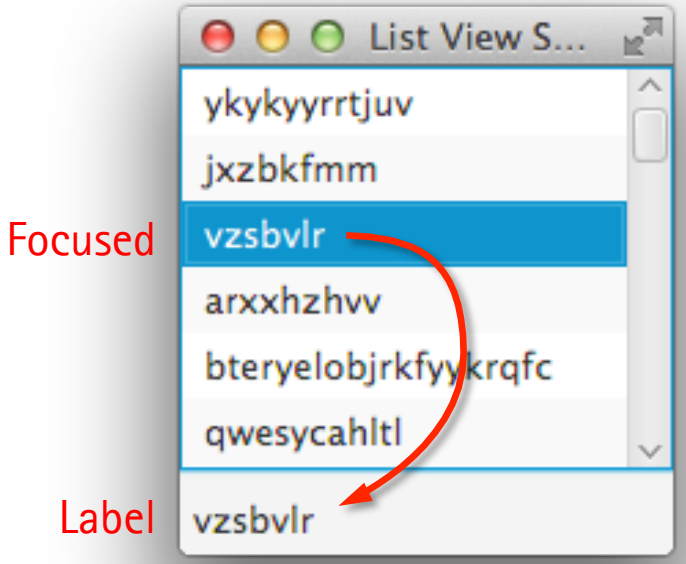
```
msm.setSelectionMode(SelectionMode.MULTIPLE);
```

Tracking State Changes: Observer Pattern

- GUI state changes on user input, external events, etc.
  - Dependencies between models, between views, constraints
  - Complex event flow
- Typical solution: Observer pattern
- Widget allows listeners to register for state changes
  - void addListener(Listener l);
  - void removeListener(Listener l);
  - List<Listener> listeners;
- Widget notifies each listener if its state changes
  - interface Listener { void notify(Object change); }
- Pros: Flexible connections between publishers and consumers
- Cons: Significant implementation overhead

Tracking State Changes in JavaFX

- JavaFX Nodes allow for fine-granular state tracking
- State of a Node (e.g., a ListView) is a set of "properties"
  - Properties allow to register listeners
  - Each property change can thus be detected and reacted upon
- Example: Properties of ListView
  - Items, selection model, focus model, context menu, height, width, hover, etc.
  - Register listener with focus model to get notified when focus changes, then update label with the focused item



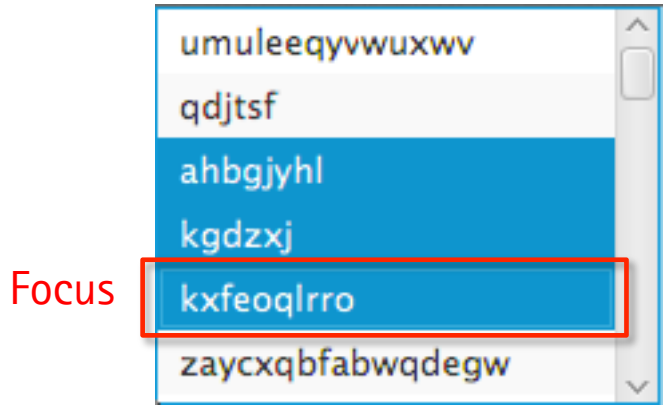
Example: Label Widget has a String Property (abbreviated)

```
class Label {
    ...
    private StringProperty text;
    public final StringProperty textProperty() {
        if (text == null) {
            text = new SimpleStringProperty(this,   "text",       "");
        }
        return text;
    }
    public final void setText(String value) { textProperty().setValue(value); }
    public final String getText() { return text == null ? "" : text.getValue(); }
    ...
}
```

Label, name, initial value

Tracking Focus Changes

- Register change and invalidation listeners with focused item property

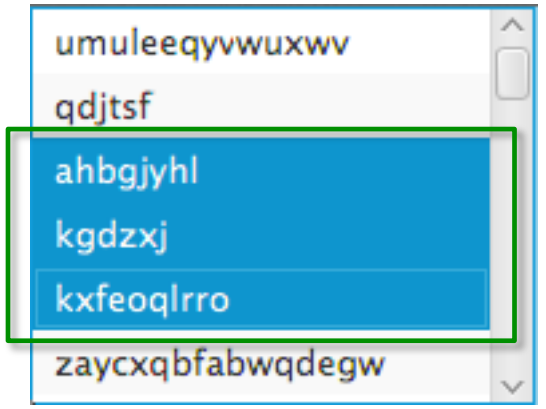



```
FocusModel<String> focusModel = listView.getFocusModel();
ReadOnlyObjectProperty<String> focusedItemProperty =
        focusModel.focusedItemProperty();
focusedItemProperty.addListener((ObservableValue<? extends String>
        observable, String oldValue, String newValue) -> {
    System.out.println(oldValue + " --> " + newValue);
});
focusedItemProperty.addListener((Observable observable) -> {
    System.out.println(focusModel.getFocusedItem());
});
```

change listener

invalidation listener

Tracking Selection Changes



- Register change and invalidation listeners with selected item(s) property

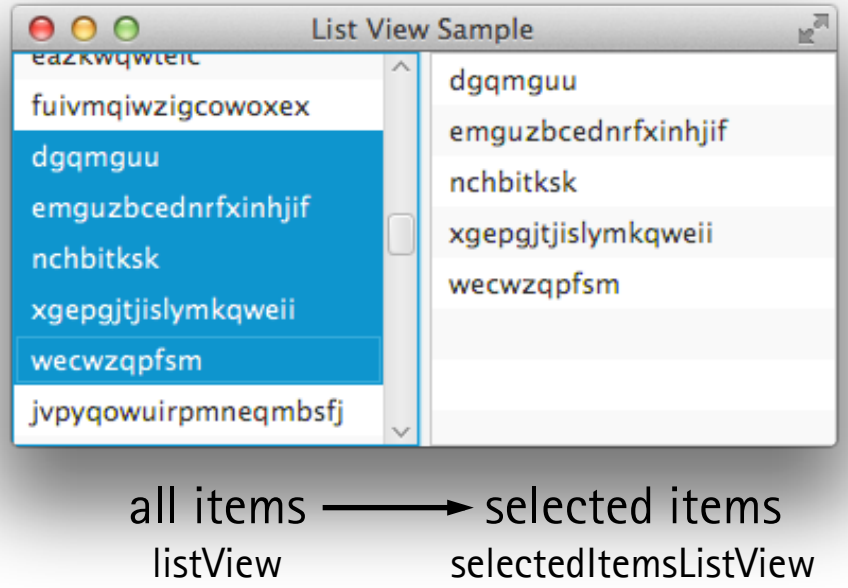
```
MultipleSelectionModel<String> selectionModel = listView.getSelectionModel();
selectionModel.setSelectionMode(SelectionMode.MULTIPLE);
ObservableList<String> selectedItems = selectionModel.getSelectedItems();
selectedItems.addListener((ListChangeListener.Change<? extends String> c) -> {
    System.out.println("change: " + c);
    show(selectedItems);
});
selectedItems.addListener((Observable observable) -> {
    show(selectedItems);
});
```

change listener

invalidation listener

Tracking Data Changes

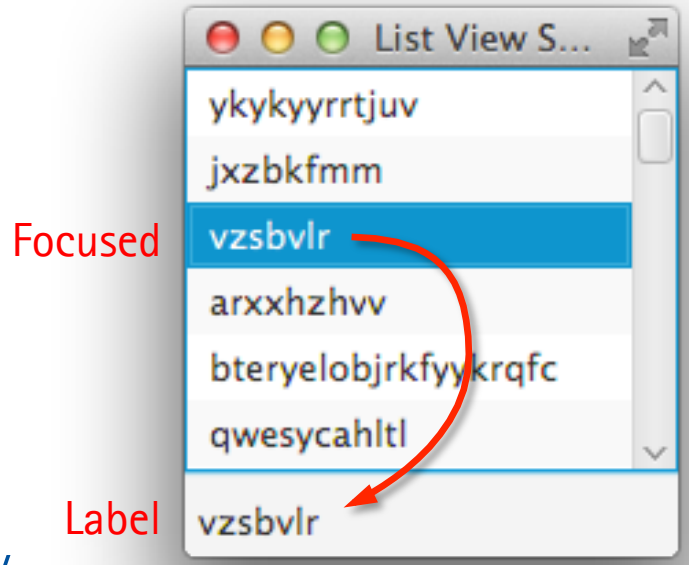
- List views automatically track changes in their data model
  - Data models are observable lists



- Example: List on the right shows the items selected on the left

```
ListView<String> listView = new ListView(generateItems(100));
MultipleSelectionModel<String> selectionModel = listView.getSelectionModel();
selectionModel.setSelectionMode(SelectionMode.MULTIPLE);
ObservableList<String> selectedItems = selectionModel.getSelectedItems();
ListView<String> selectedItemsListView = new ListView(selectedItems);
HBox root = new HBox(5, listView, selectedItemsListView);
```

Property Binding



- A property may be "bound to" another property
  - Its value then is linked to the other property

- Example: A label that always shows the focused list item
  - label.textProperty().bind(focusModel.focusedItemProperty());
  - Label's value is now "bound to" the list's focused item
  - Value of label's text property is kept "up to date" with focused item
  - Internally registers a change listener with the focus model

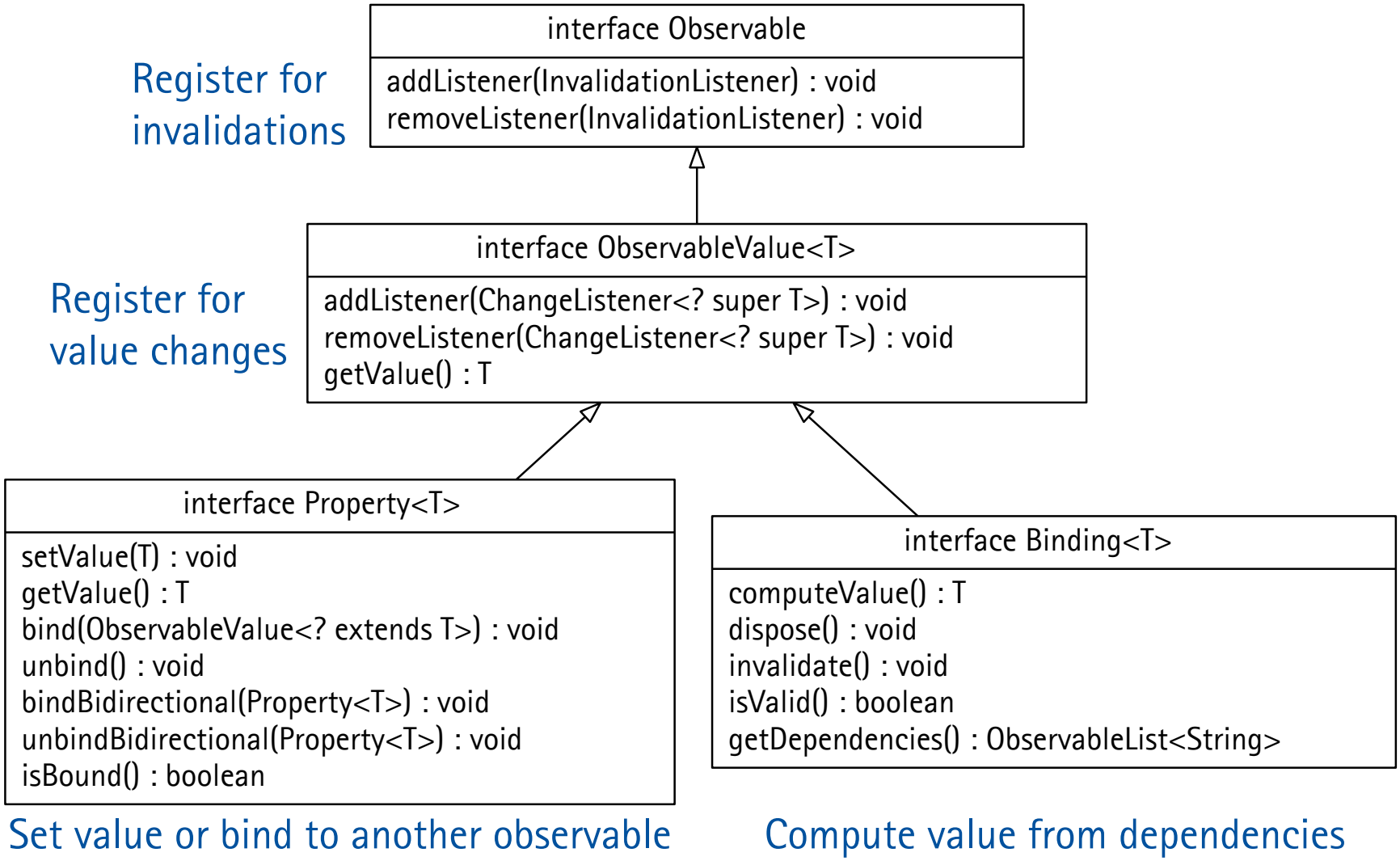
Properties

- A property either encapsulates a value
  or is bound to an observable value

- A property publishes change/invalidation notifications
  - Register listeners with a property to receive notifications
  - Change notification: Provides old and new value
  - Invalidation notification: No new value computed yet, only marked invalid

- A property is part of another object, a "bean"
  - Example: textProperty's bean is the label
- A property has a name string
  - Example: "text"

Observable Types
(abbreviated)



Interface Observable

- Wraps content and allows to observe it for invalidations
- Registering for changes
  - void addListener(InvalidationListener listener);
  - void removeListener(InvalidationListener listener);
- Invalidation listeners

```
public interface InvalidationListener {
    void invalidated(Observable observable);
}
```

- Lazy evaluation
  - Content may be marked as invalid,
    but not immediately recomputed after a change
  - Recomputed the next time the content is requested

Interface ObservableValue<T>

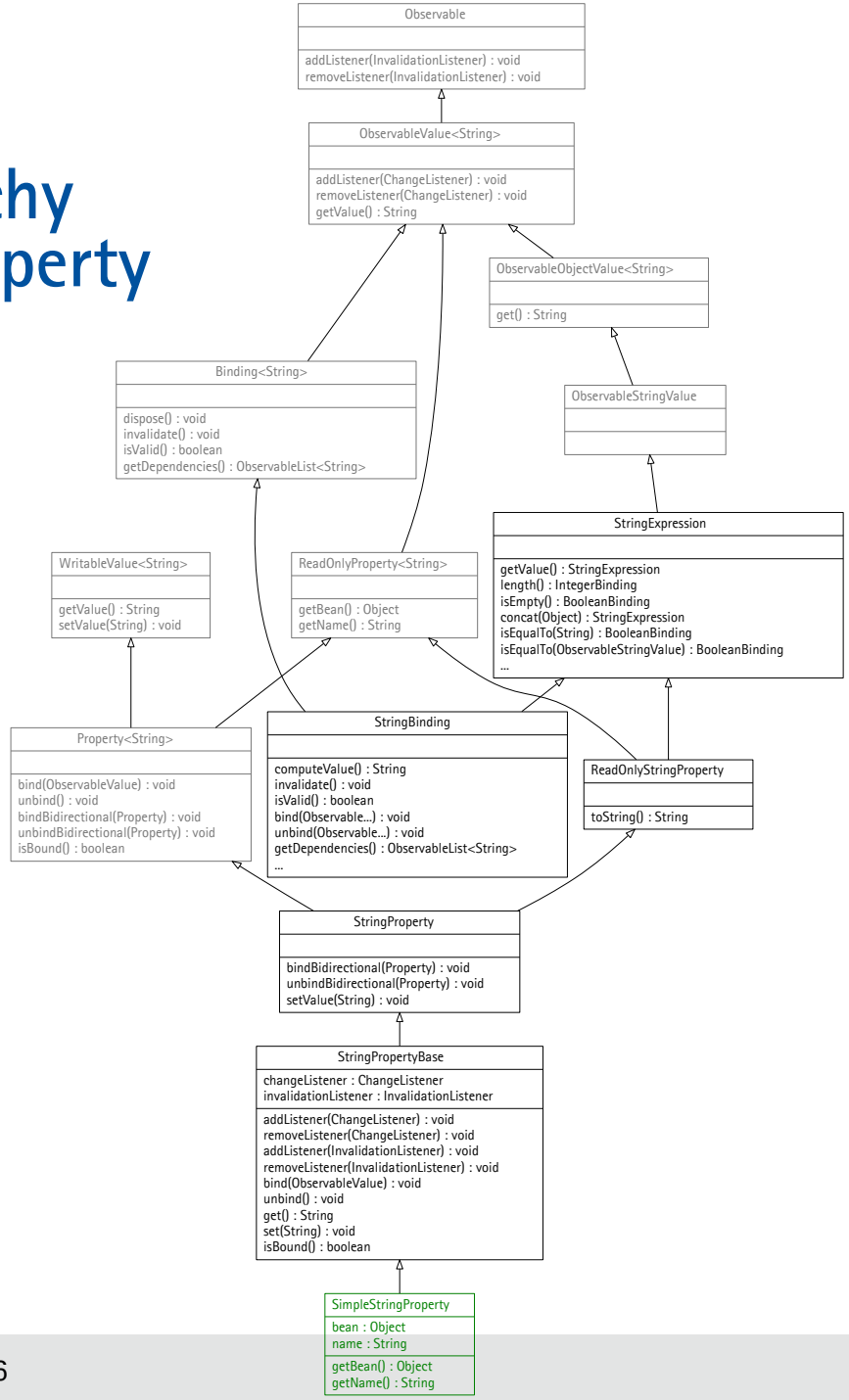
- Wraps a variable of type T
- Allows to observe changes of the variable
- Getting the value
  - T getValue();
- Registering for value changes
  - void addListener(ChangeListener<? super T> listener);
  - void removeListener(ChangeListener<? super T> listener);
- Change listeners

```
public interface ChangeListener<T> {
    void changed(ObservableValue<? extends T> observable,
                 T oldValue, T newValue);
}
```

Example: Simple String Property (abbreviated)

```
public class SimpleStringProperty extends ... {
    private String value; // initial/unbound value
    private ObservableValue<? extends String> observable; // bound to, if not null
    public SimpleStringProperty(Object bean, String name, String initialValue) {...}
    public void addListener(InvalidationListener listener) {...}
    public void addListener(ChangeListener<? super String> listener) {...}
    public String get() {...}
    public void set(String newValue) {...} // only if unbound
    public void bind(ObservableValue<? extends String> newObservable) {
        ...
        observable = newObservable;
        observable.addListener(this);
        markInvalid();
} }
```

Inheritance Hierarchy of SimpleStringProperty



Observable Lists

- interface javafx.collections.ObservableList extends java.util.List and implements javafx.beans.Observable
  - Observable lists allow adding ListChangeListeners
- ListChangeListener.Change represents a change made to an ObservableList

- class FXCollections provides static factory methods for creating observable data structures
  - Lists, maps, sets

JavaFX Script (obsolete), bind keyword

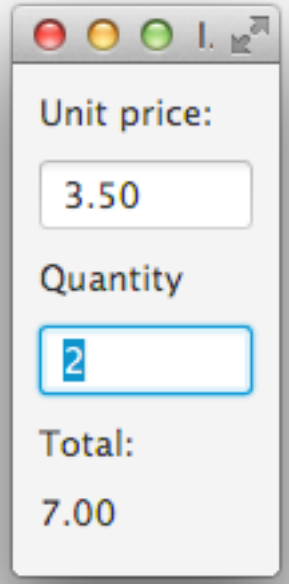
- Could be simpler, if Java had a "bind" keyword
- Variable semantics
  - Java: Variables only change when explicitly set, no notion of time-varying values/events/signals
  - Required: Variable can change on events, expressions are continuously evaluated, language support for setting up the structure of the dataflow
- Example: JavaFX Script (obsolete, but valuable for illustrating the idea)

```
var priceInput: TextBox;
var quantityInput: TextBox;
var price: Double = bind new Double(priceInput.text);
var quantity: Double = bind new Double(quantityInput.text);
var total: Double = bind price * quantity;
var total: Label { text: bind "Total: ${total}"  };
```

bind: variable tracks changes of its source

Example: Invoice Calculator, Static Structure

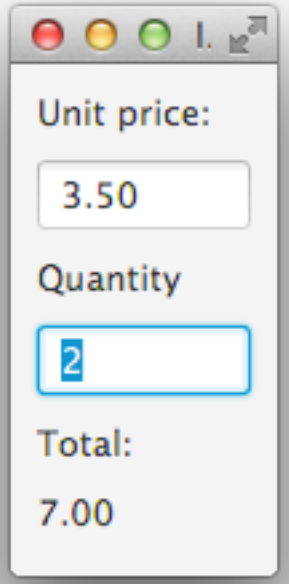
```
public void start(Stage stage) {
    Label priceLabel = new Label("Unit price:");
    TextField price = new TextField("0.00");
    Label quantityLabel = new Label("Quantity");
    TextField quantity = new TextField("0.00");
    Label totalLabel = new Label("Total:");
    Label total = new Label("0.00");
    VBox root = new VBox(10, priceLabel, unitPrice,
            quantityLabel, quantity, totalLabel, total);
    root.setPadding(new Insets(10));
    Scene scene = new Scene(root, 100, 200);
    stage.setTitle("Invoice"); stage.setScene(scene); stage.show();
}
```



simple

Example: Invoice Calculator, Listeners

```
price.textProperty().addListener(new ChangeListener<String>() {
    public void changed(ObservableValue<? extends String> o,
        String oldValue, String newValue) {
        try {
            double price = Double.parseDouble(newValue);
            double count = Double.parseDouble(quantity.getText());
            total.setText(String.valueOf(price * count));
        } catch (NumberFormatException ex) {
            total.setText("?");
        }
    }
});
quantity.textProperty().addListener( ... as above ... );
```



compli-
cated

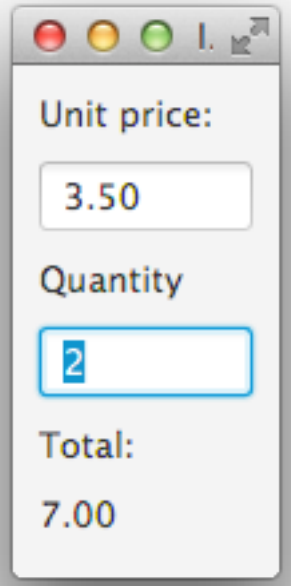
Example: Invoice Calculator, Bindings

function to compute the result

```
Callable<String> func = new Callable<String>() {
    public String call() throws Exception { return ...; }
};
StringBinding sb = Bindings.createStringBinding(
    func, price.textProperty(), quantity.textProperty());
```

dependencies

```
total.textProperty().bind(sb);
```



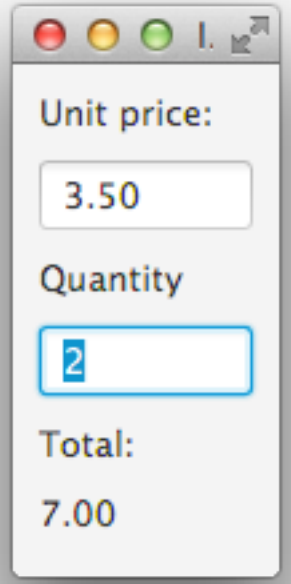
- Binding calculates value that depends on one or more sources
- Binding observes its dependencies for changes
- Updates its value automatically
- Binding<T> implements ObservableValue<T> (so Listeners may register)

Example: Invoice Calculator, Bindings

```
total.textProperty().bind(Bindings.createStringBinding(() -> {
    try {
        double p = Double.parseDouble(price.getText());
        double q = Double.parseDouble(quantity.getText());
        return String.valueOf(p * q);
    } catch (NumberFormatException ex) {
        return "?";
    }
}, price.textProperty(), quantity.textProperty()));
```

function to compute the result

dependencies



Reactive Programming

- Paradigm with focus on expressing dataflow
- Automatically propagate changes through dataflow
- Imperative programming
  - *a* := *b* + *c*; // pseudo code
  - assigns *a* when the expression is evaluated
  - subsequent changes of *b* or *c* have no effect on *a*
- Reactive programming
  - *a* <- *b* + *c*; // pseudo code
  - establishes a dataflow from *b* and *c* towards *a*
  - *a* is recomputed (+) when *b* or *c* change, *a* is always kept "up to date"
  - Example: Electrical connection through or-gate
  - Example: Spreadsheet cell referring to other cells

Reactive Programming

- Reactive programming
  - *a* <- *b* + *c*; // pseudo code
  - *a* is dependent on *b* and *c*
  - Dependency graph
  - Recomputation (+) automatically happens as *b* or *c* change

Reactive Programming and User Interfaces

- UI needs to react to and coordinate events
  - Mouse clicks, button presses, multitouch gestures, received data, etc.
- Control flow is driven by events (inverted control)
  - Difficult to predict or control events
  - State change propagation to dependencies is complex (with listeners)
- Event listeners, callbacks
  - No return value, side-effects
  - "Callback hell"
- Analysis of Adobe desktop applications
  - 33% of the code does event handling
  - 50% of reported bugs exist in event handling code

Bainomugisha et al. A survey on reactive programming. ACM Comput. Surv. 45, 4, Article 52 (August 2013).

Reactive Programming and User Interfaces

- Automatically manage the flow of time and data dependencies
- Embedding a spreadsheet-like model in programming languages
- Opportunity for dedicated language abstractions
  - Representing time-varying values
  - Handling events: event propagation, dependencies
  - Declarative specification of event chains
  - Management of structure of dataflow, composability
  - Goals: clarity, ease of construction, clean semantics

Bainomugisha et al. A survey on reactive programming. ACM Comput. Surv. 45, 4, Article 52 (August 2013).

Reactive Programming: Basic Abstractions

- Behaviors/signals: Continuous time-varying values
  - Continuously change
  - Example: time
- Events: Streams of discrete value changes
  - Occur at discrete points in time
  - Example: button press
- Behaviors/signals and events are composable to create reactive expressions
  - Combinators: combining or filtering events
  - Examples: merge, filter

ReactFX

- Reactive programming for JavaFX
- Event stream: emits events (values, changes, invalidations)
  - ObservableValues may be sources of event streams
- Subscribe to an event stream to receive notifications
- Event streams may be composed
  - filter: only emit events that match a predicate
  - map: compute a function on each event, emit the result
  - merge: combine multiple event streams into one
  - combine-map: create new stream by computing a function on input streams
  - etc.
- Convert a stream to a binding
- https://github.com/TomasMikula/ReactFX

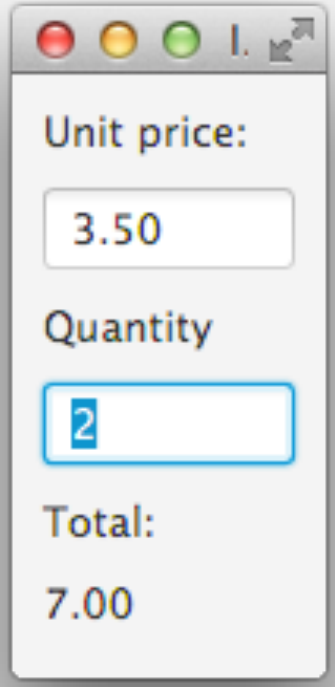
Example: Invoice Calculator, ReactFX Event Streams

- Back to Invoice Calculator,
  but with reactive programming

- Text fields generate a stream of strings:

```
EventStream<String> priceTextStream =
    EventStreams.valuesOf(price.textProperty());

EventStream<String> quanTextStream =
    EventStreams.valuesOf(quantity.textProperty());
```



(Note: Examples use ReactFX version 1.4.1)

Example: Invoice Calculator, ReactFX Event Streams

- Convert stream of Strings to stream of Doubles

```
Function<String, Double> stringToDouble = s -> {
    try {
        return Double.parseDouble(s);
    } catch (NumberFormatException ex) {
        return Double.NaN;
    }
};

EventStream<Double> priceStream = priceTextStream.map(stringToDouble);
EventStream<Double> quantityStream = quanTextStream.map(stringToDouble);
```

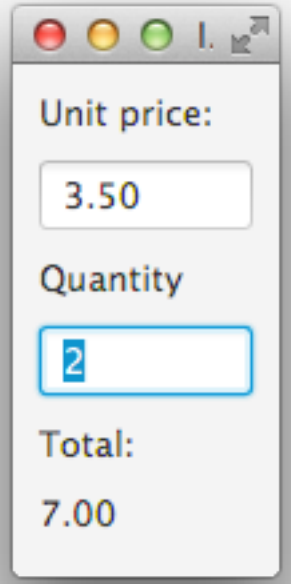
Example: Invoice Calculator, ReactFX Event Streams

- Combine streams and compute result (price * quantity):

```
EventStream<Double> totalStream = EventStreams
    .combine(priceStream, quantityStream)
    .map((p, q) -> p * q);
```

- Bind to JavaFX property

```
EventStream<String> totalTextStream =
    totalStream.map(t -> String.valueOf(t));
total.textProperty().bind(totalTextStream.toBinding("0.0"));
```

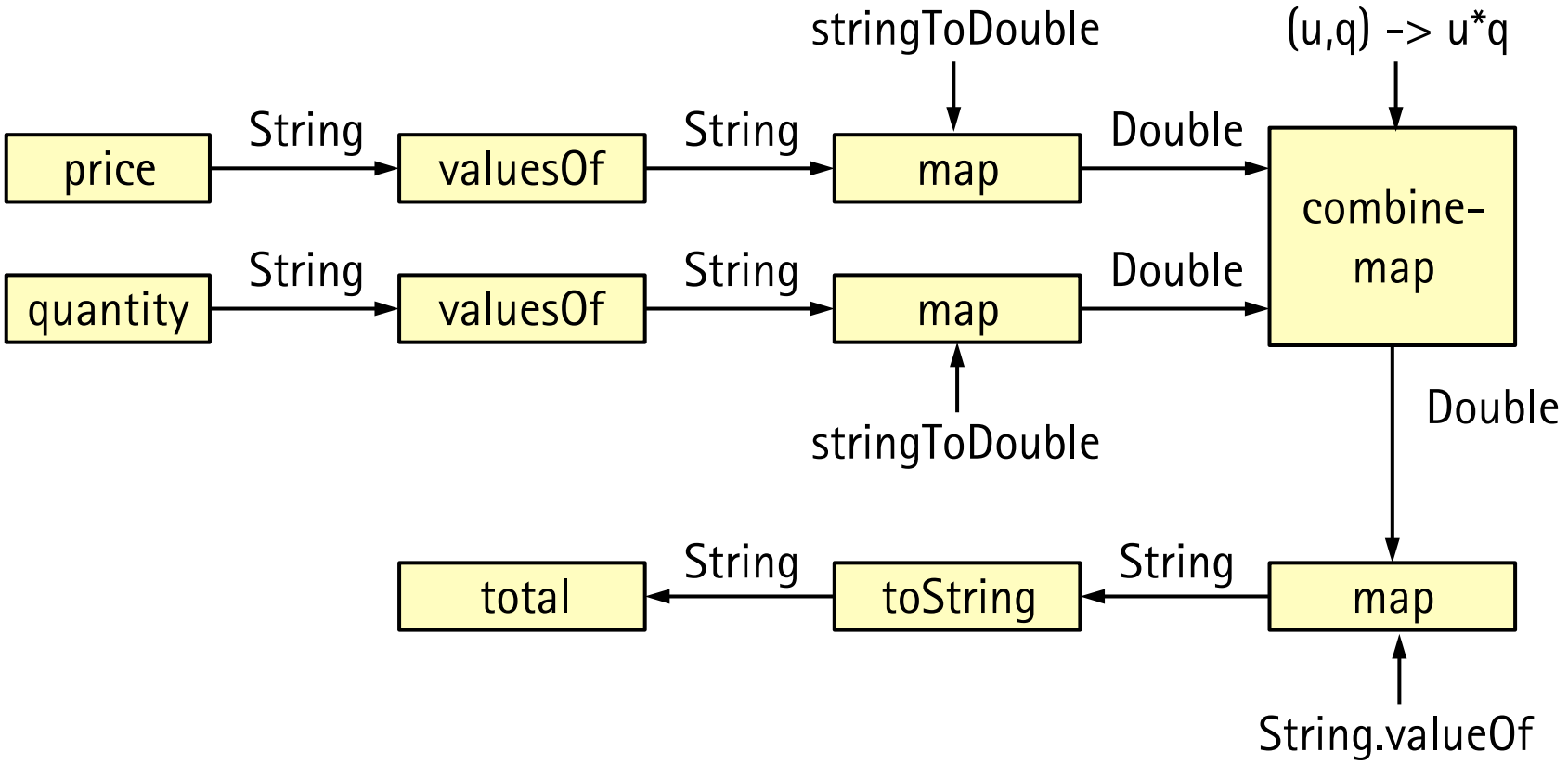


Example: Invoice Calculator, ReactFX Event Streams

```
EventStream<Double> priceStream = EventStreams
        .valuesOf(price.textProperty())
        .map(stringToDouble);
EventStream<Double> quantityStream = EventStreams
        .valuesOf(quantity.textProperty())
        .map(stringToDouble);
EventStream<Double> totalStream = EventStreams
        .combine(priceStream, quantityStream)
        .map((p, q) -> p * q);
total.textProperty().bind(totalStream
        .map(t -> String.valueOf(t))
        .toBinding("0.0"));
```

Example: Invoice Calculator, ReactFX Event Streams

- Streams explicitly describe the dataflow



Event Streams with JavaFX ObservableValues

- Implement an observable value that listens to one or more other observable values and transforms these values to a new value
  - Use Java Generics for type safety
  - Use Java8 Lambdas to implement the transformation

- ```
  static <A, R> ObservableValue<R> map(
      ObservableValue<A> ova, Function1<A, R> f);
  ```
  - map is a generic, static method
  - ova produces values of type A
  - f maps a value of type A to a value of type R

- Combine such observable values to build an event stream
  ```

Functional Interfaces

- 1-argument function: Take an A, return an R

```
public interface Function1<A, R> {
    public R apply(A a);
}
```

- 2-argument function: Take an A and a B, return an R

```
public interface Function2<A, B, R> {
    public R apply(A a, B b);
}
```

- 3-argument function: Take an A and a B and a C, return an R

...

Event Streams with JavaFX ObservableValues

- `static <A, B, R> ObservableValue<R> map2(`
  `ObservableValue<A> ova,`
  `ObservableValue<B> ovb,`
  `Function2<A, B, R> f);`
  - map2 is a generic, static method
  - ova produces values of type A
  - ovb produces values of type B
  - f maps a value of type A and a value of type B to a value of type R

- `static <A,B,C,R> ObservableValue<R> map3(...);`

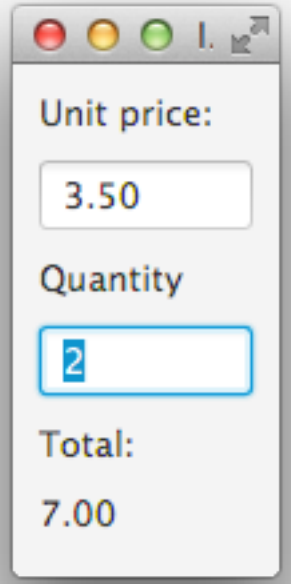
Event Streams with JavaFX ObservableValues: Example

```
import static invoice2.Lifting.*;

public class Invoice2 extends Application {

   @Override
   public void start(Stage stage) {
      // structure...
      // behavior...
   }

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



Event Streams with JavaFX ObservableValues: Example

```
// behavior

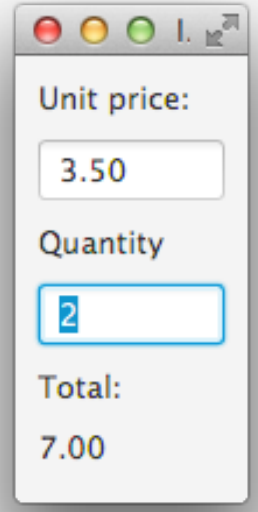
ObservableValue<Double> prices =
     map(price.textProperty(), Double::parseDouble);

ObservableValue<Double> quants =
     map(quantity.textProperty(), Double::parseDouble);

ObservableValue<Double> totals =
     map2(prices, quants, (p, q) -> p * q);

ObservableValue<String> totalsString =
     map(totals, String::valueOf);

total.textProperty().bind(totalsString);
```



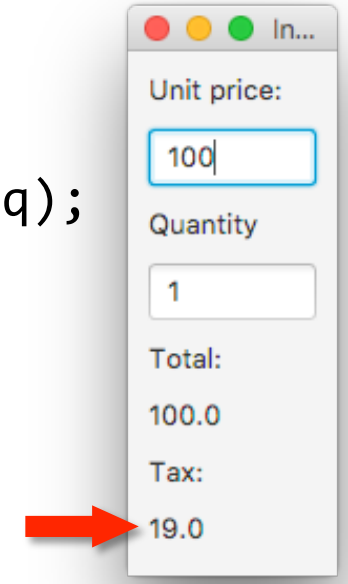
Event Streams with JavaFX ObservableValues: Example

- Setting up the dataflow…

```
ObservableValue<Double> taxes =
    map2(prices, quants, (p, q) -> 0.19 * p * q);

ObservableValue<String> taxesString =
    map(taxes, String::valueOf);

tax.textProperty().bind(taxesString);
```



- …may be written more compactly as

```
tax.textProperty().bind(map(
    map2(prices, quants, (p, q) -> 0.19 * p * q),
    String::valueOf));
```

Events Streams with ObservableValues: Implementation

```
public class Lifting {
    private static class OV1<A, R> implements ObservableValue<R> { ... }
    private static class OV2<A, B, R> implements ObservableValue<R> { ... }
    public static <A, R> ObservableValue<R> map(
        ObservableValue<A> ova, Function1<A, R> f)
    {
        OV1<A, R> ov1 = new OV1(ova, f);
        return ov1;
    }
    public static <A, B, R> ObservableValue<R> map2(
        ObservableValue<A> ova, ObservableValue<B> ovb, Function2<A, B, R> f)
    {
        OV2<A, B, R> ov2 = new OV2(ova, ovb, f);
        return ov2;
    }
}
```

Stud.IP:
Invoice2.zip

Events Streams with ObservableValues: Implementation

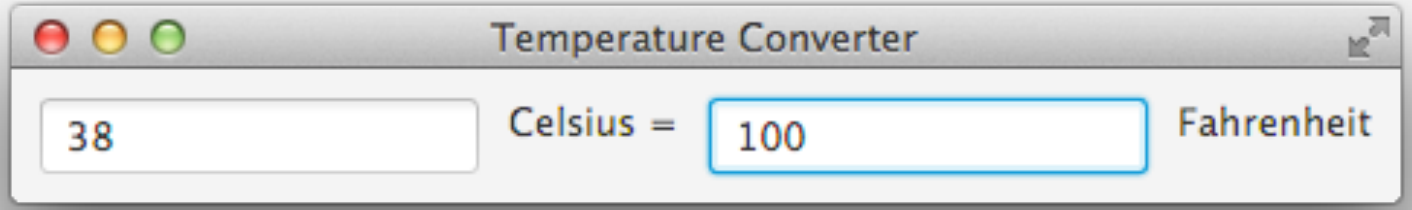
```
private static class OV1<A, R> implements ObservableValue<R> {
    private final ObservableValue<A> ovs;
    private final Function1<A, R> f;
    private final ArrayList<ChangeListener<? super R>> changeListeners;
    private final ArrayList<InvalidationListener> invalidationListeners;
    public OV1(ObservableValue<A> ovs, Function1<A, R> f) { ... }
    public R getValue() {
        A s = ovs.getValue();
        R r = f.apply(s);
        return r;
    }
    public void addListener(ChangeListener<? super R> listener) { ... }
    public void removeListener(ChangeListener<? super R> listener) { ... }
    public void addListener(InvalidationListener listener) { ... }
    public void removeListener(InvalidationListener listener) { ... }
}
```

Events Streams with ObservableValues: Implementation

```
public OV1(ObservableValue<A> ovs, Function1<A, R> f) {
    this.ovs = ovs;
    this.f = f;
    this.changeListeners = new ArrayList();
    this.invalidationListeners = new ArrayList();
    ovs.addListener(new ChangeListener<A>() {
        public void changed(ObservableValue<? extends A> o, A sOld, A sNew) {
            R rOld = f.apply(sOld);
            R rNew = f.apply(sNew);
            for (ChangeListener l : changeListeners) {
                l.changed(OV1.this, rOld, rNew);
            }
        }
    });
    ovs.addListener(new InvalidationListener() { ... });
}
```

Bidirectional Bindings Example: Temperature Converter

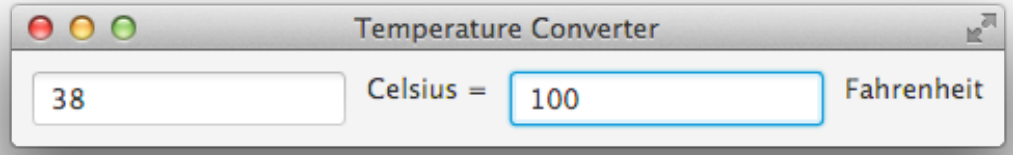
```
TextField celsius = new TextField();
TextField fahrenheit = new TextField();
HBox root = new HBox(10, celsius,
                         new Label("Celsius ="),
                         fahrenheit,
                         new Label("Fahrenheit"));
```



Author: E. Kiss

Bidirectional Bindings Example: Temperature Converter

- Changing either field modifies the other




```
celsius.textProperty().bindBidirectional(fahrenheit.textProperty(),
    new StringConverter<String>() {
        public String toString(String fahrenheit) {
            return isNumeric(fahrenheit) ? fToC(fahrenheit) : celsius.getText();
        }
        public String fromString(String celsius) {
            return isNumeric(celsius) ? cToF(celsius) : fahrenheit.getText();
        }
    });
```

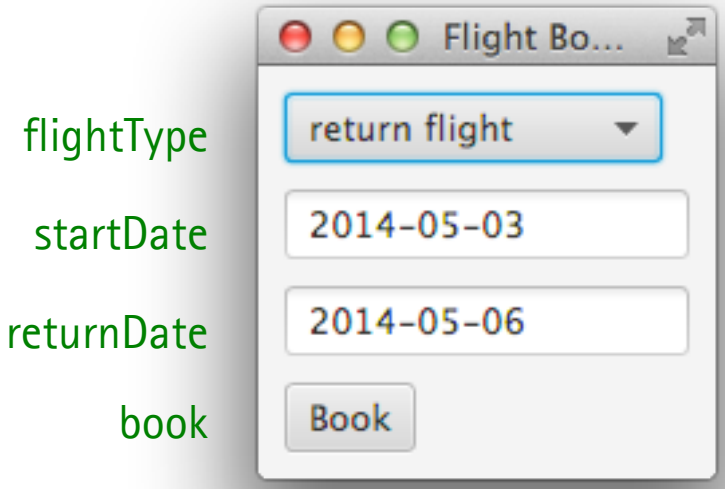
°F ⟶ °C

°F ⟵ °C

Author: E. Kiss

Complex Widget Constraints

- Expressing dependencies between widgets




```
ComboBox<String> flightType = new ComboBox<>();
flightType.getItems().addAll("one-way flight", "return flight");
flightType.setValue("one-way flight");
TextField startDate = new TextField(dateToString(LocalDate.now()));
TextField returnDate = new TextField(dateToString(LocalDate.now()));
Button book = new Button("Book");

VBox root = new VBox(10, flightType, startDate, returnDate, book);
root.setPadding(new Insets(10));
```

Author: E. Kiss

Complex Widget Constraints

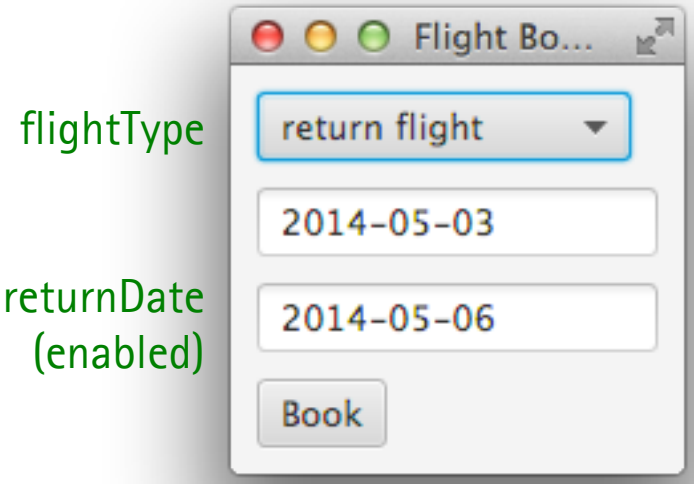
- Only enable return date if it is a return flight

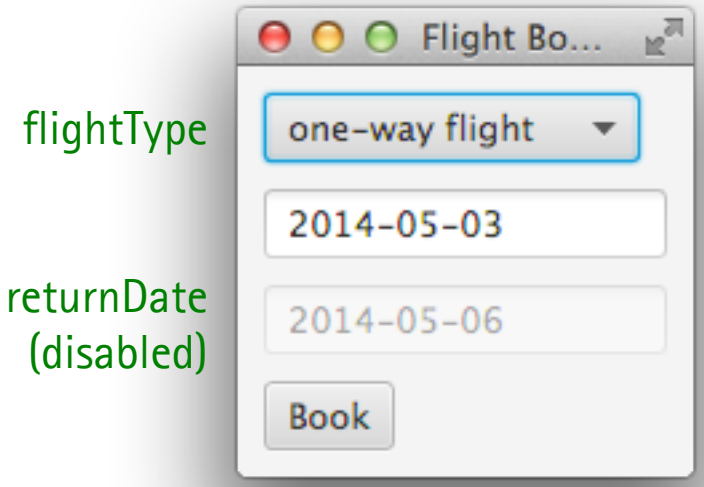
BooleanProperty

```
returnDate.disableProperty().bind(
    flightType.valueProperty().isEqualTo("one-way flight"));
```

StringProperty

BooleanProperty

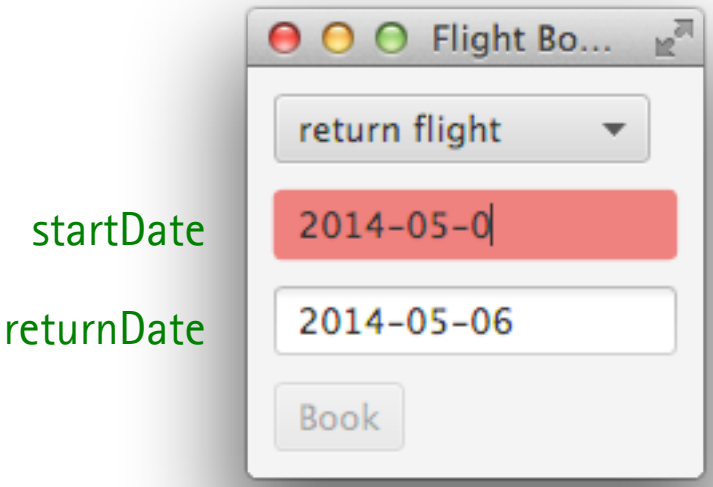




Author: E. Kiss

Complex Widget Constraints

- Change the background color if the input value is not a date



startDate

returnDate

```
startDate.textProperty().addListener((v, o, n) ->
    startDate.setStyle(isDateString(n) ?
        "" : "-fx-background-color: lightcoral"));

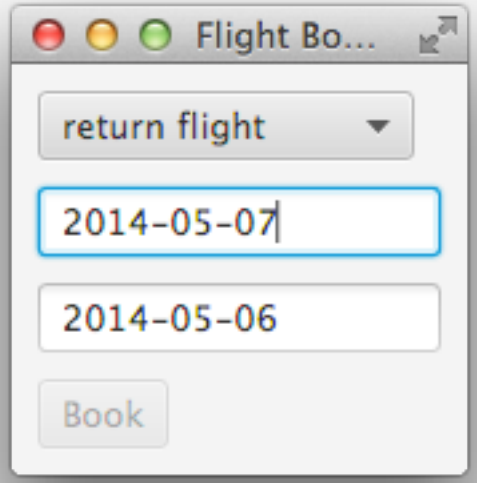
returnDate.textProperty().addListener((v, o, n) ->
    returnDate.setStyle(isDateString(n) ?
        "" : "-fx-background-color: lightcoral"));
```

CSS for styling widgets

Author: E. Kiss

Complex Widget Constraints

- Only enable the Book button if the dates are valid and the return date is not before the start date



book

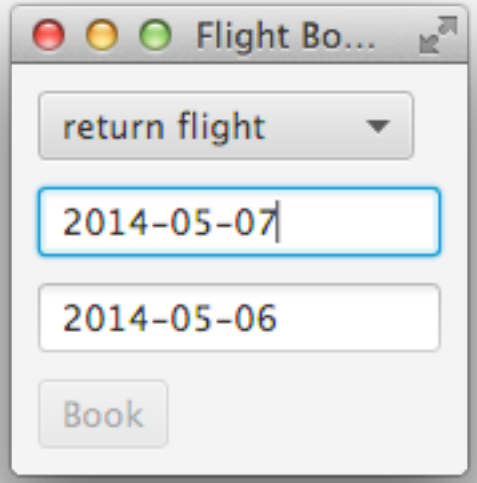
```
ChangeListener bookEnabledAction = (v, o, n) -> {
    switch (flightType.getValue()) {
        case "one-way flight":
            book.setDisable(!isDateString(startDate.getText()));
            break;
        case "return flight":
            book.setDisable(...); // next slide
            break;
} };
```

Author: E. Kiss

Complex Widget Constraints

```
book.setDisable(
        !isDateString(startDate.getText())
    || !isDateString(returnDate.getText())
    || stringToDate(startDate.getText())
        .compareTo(stringToDate(
        returnDate.getText())) > 0);
```

book



...

```
flightType.valueProperty().addListener(bookEnabledAction);
startDate.textProperty().addListener(bookEnabledAction);
returnDate.textProperty().addListener(bookEnabledAction);
```

if either property changes, Book button state may change

Author: E. Kiss

JavaFX Threads

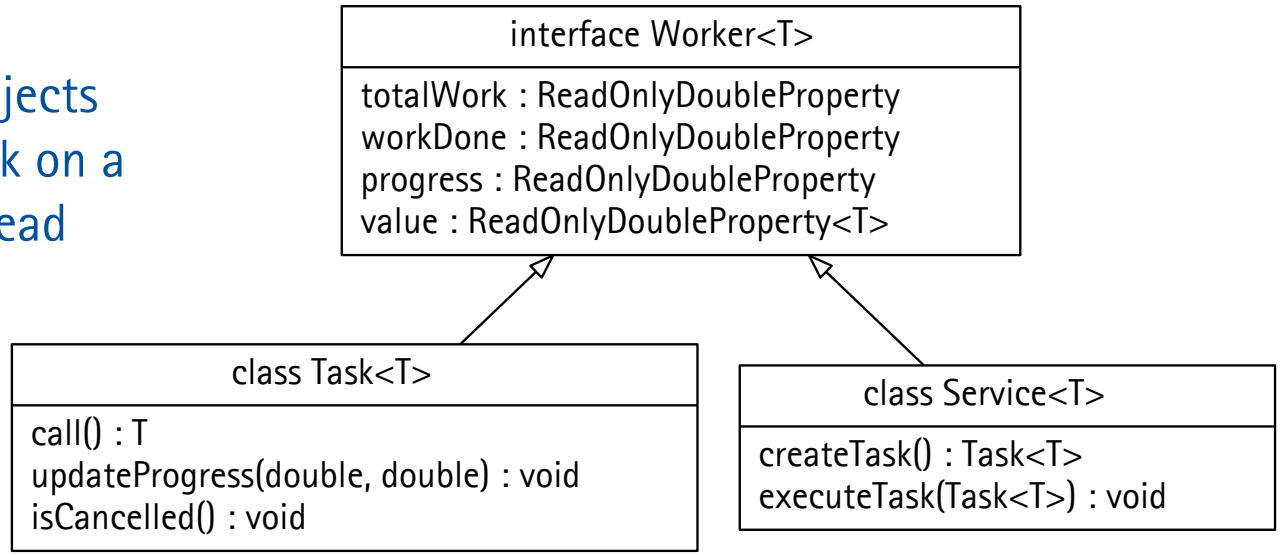
- JavaFX Application thread
  - Primary thread
  - Event dispatch
- Render thread
  - 60 fps
  - Batch and execute events on a pulse
- Worker threads
  - Long-running tasks
  - Programmer-defined

Concurrency

- UI has to stay responsive
- Long-running tasks typically run on separate threads
- Start worker thread using Java API (Thread, Runnable, etc.)
  - package java.util.concurrent
  - Thread t = new Thread(task); // task implements Runnable
  - t.setDaemon(true); // daemon threads don't keep the VM alive
  - t.start(); // call Runnable.run() on the new thread
- Only manipulate JavaFX scene graph from main thread
  - JavaFX Application Thread
- Communication between worker threads and main thread
  - Synchronization between threads
    - package javafx.concurrent

Concurrency API

- Worker
  - Interface for objects performing work on a background thread
- Task
  - Executes work
  - Implement call method
  - Properties of Worker may be invoked from the main thread
- Service
  - Executes tasks
  - Register completion listeners



Modifying Scene Graph from Worker Thread

```
final Group group = new Group();
Task<Void> task = new Task<Void>() { // javafx.concurrent.Task (impl. Runnable)
    protected Void call() throws Exception {
        for (int i = 0; i < 100; i++) {
            updateProgress(i, 100); // update Task's progress property (main thread)
            if (isCancelled()) break; // regularly check if this thread should end
            final Rectangle r = new Rectangle(10, 10, Color.RED);
            r.setX(11 * i);
            Platform.runLater(new Runnable() { // run runnable on main thread
                public void run() {
                    group.getChildren().add(r);
                }
            });
            Thread.sleep(300); // sleep 300 ms
        } return null; } };
```

Updating the UI as Data Arrives

- Typical case when data is loaded from network or disk
- Items are added to long list or table one by one
- Loading is performed in the background thread
- Updating the UI and the UI's model is performed in the JavaFX Application thread

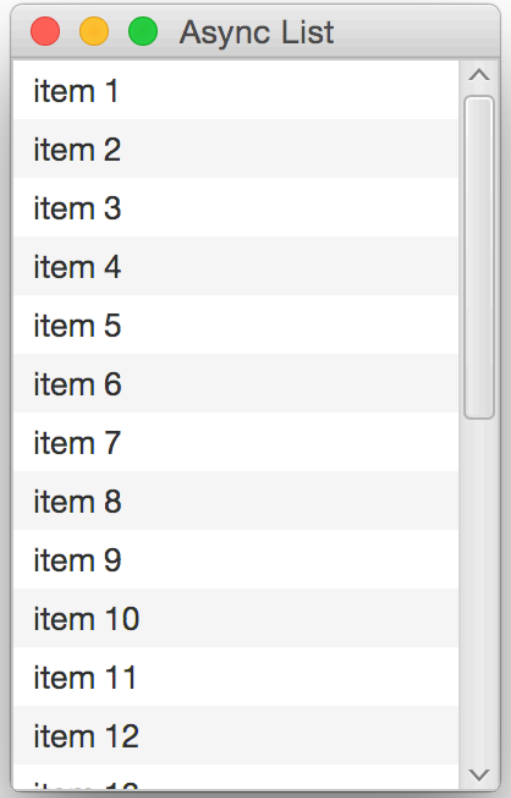
- Example

```
ObservableList<String> items = FXCollections.observableArrayList();
ListView<String> listView = new ListView<>(items);
```

  - items will be added as new data arrives

Updating the UI as Data Arrives

```
public class AsyncList extends Application {
    ObservableList<String> items = FXCollections.observableArrayList();
    @Override public void start(Stage stage) {
        ListView<String> listView = new ListView<>(items);
        Scene scene = new Scene(listView, 200, 300);
        stage.setTitle("Async List");
        stage.setScene(scene);
        stage.show();
        // ← load data in background thread
    }
    public static void main(String[] args) {
        launch(args);
} }
```



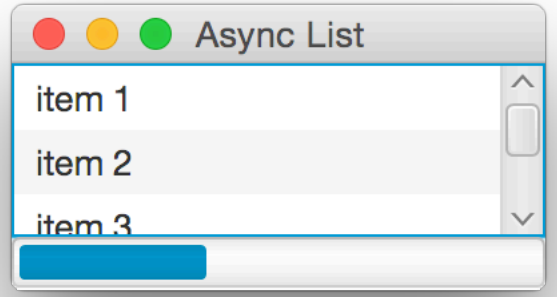
Updating the UI as Data Arrives

```
// load data in background thread
Task<Void> listLoadingTask; // task will load items
listLoadingTask = new Task<Void>() { // javafx.concurrent.Task (impl. Runnable)
@Override protected Void call() throws Exception {
    for (int i = 1; i <= 100; i++) {
        updateProgress(i, 100); // update Task's progress property (main thread)
        if (isCancelled()) {
            break; // regularly check if this thread should end
        }
        final int _i = i;
        Platform.runLater(() -> { // run on main thread
            items.add("item " + _i); // needs a final variable
        });
```

Updating the UI as Data Arrives

```
// load data in background thread (continued)
            try {
                Thread.sleep(300); // sleep for 300 ms
            } catch (InterruptedException ex) {}
        }
        return null;
    }
};
Thread thread = new Thread(listLoadingTask);
thread.setDaemon(true); // thread does not keep VM alive
thread.start(); // actually start the task
}
```

Observing Task Progress



- Bind scene graph node to task property

```
Task<Void> task = new Task<Void>() { ... }
ProgressBar bar = new ProgressBar();
bar.progressProperty().bind(task.progressProperty()); // bind to task property
new Thread(task).start(); // actually start the task
```

- Task methods for state changes of task
  - succeeded
  - cancelled
  - failed
- Task management with service class
  - Initialization data for task
  - State changes

Canvas API

- Immediate mode API
  - Drawn shapes do not exist as objects in a scene graph
  - Methaphor: Drawing on a canvas
- Canvas itself is a node in the scene graph
  - Multiple canvases in scene graph to create layers
  - Order of canvases in scene graph: toBack(), toFront()
- GraphicsContext for issuing draw calls
  - Maintains a stack for the drawing state (save, restore)
- Example

```
Canvas canvas = new Canvas(300, 250);
GraphicsContext gc = canvas.getGraphicsContext2D();
gc.setFill(Color.GREEN);    gc.fillOval(10, 60, 30, 30);
```

GraphicsContext State

- Global Alpha
- Global Blend Operation
- Transform
- Fill Paint
- Stroke Paint
- Line Width
- Line Cap
- Line Join
- Miter Limit
- Number of Clip Paths
- Font
- Text Align
- Text Baseline
- Effect
- Fill Rule