Reactive Programming
Functional Reactive Programming
Functional Reactive Programming
1. **First-class functions**: functions that can take other functions as arguments, or return functions.

```javascript
function greet(prefix) {
  return function(msg) {
    console.log(prefix + msg);
  }
}
```
1. **First-class functions**: functions that can take other functions as arguments, or return functions.

2. **Declarative Specification**: describe *what* we want (i.e., the logic/structure) not *how* to compute it (i.e., no loops/ifs/etc.).
 Functional Programming Principles

1. **First-class functions**: functions that can take other functions as arguments, or return functions.

2. **Declarative Specification**: describe what we want (i.e., the logic/structure) not how to compute it (i.e., no loops/ifs/etc.).

3. **No* side effects**: no* modifications to state outside the local scope.

Side effects make debugging state difficult. What code paths modify the state? Does the state depend on history?

```javascript
let allTeas;
get("/teas", function(text) {
    allTeas = JSON.parse(text);
});
```
1. **First-class functions**: functions that can take other functions as arguments, or return functions.

2. **Declarative Specification**: describe what we want (i.e., the logic/structure) not how to compute it (i.e., no loops/ifs/etc.).

3. **No* side effects**: no* modifications to state outside the local scope.

4. **Pure Functions**: cause no side effects, returns the same value for the same arguments.

Purity is related to idempotency, but an idempotent *impure* function can have idempotent side effects

```javascript
function setStyle(sel, prop, val) {
    let elem = document.querySelector(sel);
    elem.style[prop] = val;
}
```
Functional Programming Principles

1. **First-class functions**: functions that can take other functions as arguments, or return functions.

2. **Declarative Specification**: describe what we want (i.e., the logic/structure) not how to compute it (i.e., no loops/ifs/etc.).

3. **No side effects**: no modifications to state outside the local scope.

4. **Pure Functions**: cause no side effects, returns the same value for the same arguments.

5. **Immutable Objects**: cannot change an object after it's been created.

```javascript
b = ["foo", "bar"];  // Correct
× b[2] = "baz";  // Incorrect
× b.append("wibble");  // Incorrect
```
1. **First-class functions**: functions that can take other functions as arguments, or return functions.

2. **Declarative Specification**: describe what we want (i.e., the logic/structure) not how to compute it (i.e., no loops/ifs/etc.).

3. **No side effects**: no modifications to state outside the local scope.

4. **Pure Functions**: cause no side effects, returns the same value for the same arguments.

5. **Immutable Objects**: cannot change an object after it's been created.

```javascript
a = {hello: "there"};
× a.hello = "world";
× a.goodbye = "for now";
```
1. **First-class functions**: functions that can take other functions as arguments, or return functions.

2. **Declarative Specification**: describe what we want (i.e., the logic/structure) not how to compute it (i.e., no loops/ifs/etc.).

3. **No side effects**: no modifications to state outside the local scope.

4. **Pure Functions**: cause no side effects, returns the same value for the same arguments.

5. **Immutable Objects**: cannot change an object after it's been created.

```javascript
a = {hello: "there"};
Object.freeze(a);

a.hello = "world";
a.goodbye = "for now";
Silently ignored
```
In-Class Activity

1. What functional programming principles does `fireSale` violate? (~3 mins)

```javascript
let toys = [
    {name: "Woody", price: 10},
    {name: "Buzz", price: 15},
    {name: "Rex", price: 3},
    {name: "Slinky", price: 7}
];

function fireSale(minPrice, perc) {
    for (let i = 0; i < toys.length; i++) {
        if (toys[i].price < minPrice) {
            toys.splice(i, 1);
        } else {
            toys[i].price *= perc;
        }
    }
    return toys;
}
```
 Imperative code: specifying control flow (the how).
Declarative code describes the what (i.e., logic/structure) only.

```javascript
let toys = [
  {name: "Woody", price: 10},
  {name: "Buzz", price: 15},
  {name: "Rex", price: 3},
  {name: "Slinky", price: 7}
];

function fireSale(minPrice, perc) {
  for (let i = 0; i < toys.length; i++) {
    if (toys[i].price < minPrice) {
      toys.splice(i, 1);
    } else {
      toys[i].price *= perc;
    }
  }
  return toys;
}
```
Imperative code: specifying control flow (the how). Declarative code describes the what (i.e., logic/structure) only.

Why it matters? Much easier to make mistakes (like I did here!) when you explicitly manipulate control flow.

```javascript
let toys = [
    {name: "Woody", price: 10},
    {name: "Buzz", price: 15},
    {name: "Rex", price: 3},
    {name: "Slinky", price: 7}
];

function fireSale(minPrice, perc) {
    for (let i = 0; i < toys.length; i++) {
        if (toys[i].price < minPrice) {
            toys.splice(i, 1);
        } else {
            toys[i].price *= perc;
        }
    }
    return toys;
}
```
**Imperative** code: specifying control flow (the *how*). **Declarative** code describes the *what* (i.e., logic/structure) only.

Why it matters? Much easier to make mistakes (*like I did here!*) when you explicitly manipulate control flow.

```javascript
let toys = [
  {name: "Woody", price: 10},
  {name: "Buzz", price: 15},
  {name: "Rex", price: 3},
  {name: "Slinky", price: 7}];

function fireSale(minPrice, perc) {
  for (let i = 0; i < toys.length; i++) {
    if (toys[i].price < minPrice) {
      toys.splice(i--, 1);
    } else {
      toys[i].price *= perc;
    }
  }
  return toys;
}
```
Imperative code: specifying control flow (the how).
Declarative code describes the what (i.e., logic/structure) only.

```javascript
let toys = [
    {name: "Woody", price: 10},
    {name: "Buzz", price: 15},
    {name: "Rex", price: 3},
    {name: "Slinky", price: 7}
];

function fireSale(minPrice, perc) {
    for (let i = 0; i < toys.length; i++) {
        if (toys[i].price < minPrice) {
            toys.splice(i-- , 1);
        } else {
            toys[i].price *= perc;
        }
    }
    return toys;
}
```

Mutating data and side effects.
**Imperative** code: specifying control flow (the *how*).

**Declarative** code describes the *what* (i.e., logic/structure) only.

**Mutating data** and **side effects**.

**Impure** function: return value is not dependent only on the arguments. Repeatedly calling the function will produce different results!

```javascript
let toys = [
    {name: "Woody", price: 10},
    {name: "Buzz", price: 15},
    {name: "Rex", price: 3},
    {name: "Slinky", price: 7}
];

function fireSale(minPrice, perc) {
    for (let i = 0; i < toys.length; i++) {
        if (toys[i].price < minPrice) {
            toys.splice(i--, 1);
        } else {
            toys[i].price *= perc;
        }
    }
    return toys;
}
```
In-Class Activity

1. What functional programming principles does `fireSale` violate? (~3 mins).

2. Convert `fireSale` to a pure, declarative function. (~7 mins).

(a function that only keeps toys priced greater than `minPrice`, and discounts them by `perc`).

```javascript
let toys = [
  {name: "Woody", price: 10},
  {name: "Buzz", price: 15},
  {name: "Rex", price: 3},
  {name: "Slinky", price: 7}
];

function fireSale(minPrice, perc) {
  for (let i = 0; i < toys.length; i++) {
    if (toys[i].price < minPrice) {
      toys.splice(i--, 1);
    } else {
      toys[i].price *= perc;
    }
  }
  return toys;
}
```
1. **First-class functions**: functions that can take other functions as arguments, or return functions.

2. **Declarative Specification**: describe what we want (i.e., the logic/structure) not how to compute it (i.e., no loops/ifs/etc.).

3. **No* side effects**: no* modifications to state outside the local scope.

4. **Pure Functions**: cause no side effects, returns the same value for the same arguments.

5. **Immutable Objects**: cannot change an object after it's been created.
Functional Programming Principles

1. **First-class functions**: functions that can take other functions as arguments, or return functions.

2. **Declarative Specification**: describe what we want (i.e., the logic/structure) not how to compute it (i.e., no loops/ifs/etc.).

3. **No* side effects**: no* modifications to state outside the local scope.

4. **Pure Functions**: cause no side effects, returns the same value for the same arguments.

5. **Immutable Objects**: cannot change an object after it's been created.

Why bother?
Why bother?

*Reading* code is easier: state changes are better *encapsulated*.

*Writing* code is easier: no worrying about low-level details (e.g., iteration).

*Debugging & Testing* is easier: test different inputs to functions.
Functional Reactive Programming
Functional Reactive Programming
This is the old editor for Vega 2 and Vega-Lite 1. For the latest versions, use the new editor!
switch.addEventListener('change', function(event) {
  if (this.value === 'on') {
    bulb.style.backgroundColor = 'yellow';
  } else {
    bulb.style.backgroundColor = null;
  }
});
Low-level control flow specified.

```javascript
switch.addEventListener('change', function(event) {
  if (this.value === 'on') {
    bulb.style.backgroundColor = 'yellow';
  } else {
    bulb.style.backgroundColor = null;
  }
});
```

Lightbulb is modified via side effects. Difficult to know what other states the bulb can be in (e.g., other colors?).
body.addEventListener('mousemove', function(event) {
  ...
});

Event streams: an abstraction that treats events as a continuous stream of data (rather than one-off occurrences that must be captured and handled).
FRP: Operators on *Event Streams*

Filtering events out of streams based on their properties.

Debouncing or throttling the frequency of events in a stream.
Merging individual streams into a single stream with interleaved events.
let dragging = null;

elem.addEventListener('mousedown', function(event) {
  dragging = this;
});

elem.addEventListener('mouseup', function(event) {
  dragging = null;
});

document.addEventListener('mousemove', function(event) {
  // absolute positioning
  dragging.style.left = event.pageX;
  dragging.style.top = event.pageY;
});
let dragging = null;

elem.addEventListener('mousedown', function(event) {
    dragging = this;
});

elem.addEventListener('mouseup', function(event) {
    dragging = null;
});

document.addEventListener('mousemove', function(event) {
    // absolute positioning
    dragging.style.left = event.pageX;
    dragging.style.top = event.pageY;
});

Merged stream of mousedown, mouseup events

Switch operator

Stream ofmousemove events
FRP: Operators on Event Streams

```javascript
let dragging = null;

elem.addEventListener('mousedown', function(event) {
  dragging = this;
});

elem.addEventListener('mouseup', function(event) {
  dragging = null;
});

document.addEventListener('mousemove', function(event) {
  // absolute positioning
  dragging.style.left = event.pageX;
  dragging.style.top = event.pageY;
});
```

Merged stream of mousedown, mouseup events

Switch operator

Stream ofmousemove events
let dragging = null;

elem.addEventListener('mousedown', function(event) {
  dragging = this;
});

elem.addEventListener('mouseup', function(event) {
  dragging = null;
});

document.addEventListener('mousemove', function(event) {
  // absolute positioning
  dragging.style.left = event.pageX;
  dragging.style.top = event.pageY;
});

Merged stream of mousedown, mouseup events

Switch operator

Stream ofmousemove events
let dragging = null;

elem.addEventListener('mousedown', function(event) {
    dragging = this;
});

elem.addEventListener('mouseup', function(event) {
    dragging = null;
});

document.addEventListener('mousemove', function(event) {
    // absolute positioning
    dragging.style.left = event.pageX;
    dragging.style.top = event.pageY;
});

Merged stream of mousedown, mouseup events

Switch operator

Stream of mousemove events
let dragging = null;

elem.addEventListener('mousedown', function(event) {
  dragging = this;
});

elem.addEventListener('mouseup', function(event) {
  dragging = null;
});

document.addEventListener('mousemove', function(event) {
  // absolute positioning
  dragging.style.left = event.pageX;
  dragging.style.top = event.pageY;
});

Vega provides an event stream selection syntax:

`[mousedown, mouseup] > mousemove`
In practice, few libraries expose event streams to users directly. Instead, event streams are constructed in the background, and their most recent values are available as *reactive variables*. 
<table>
<thead>
<tr>
<th></th>
<th>Oct</th>
<th>Nov</th>
<th>Dec</th>
<th>Total</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Everyday</strong></td>
<td>Monthly totals:</td>
<td>$0</td>
<td>$0</td>
<td>$0</td>
<td>$0</td>
</tr>
<tr>
<td>46</td>
<td>Groceries</td>
<td></td>
<td></td>
<td></td>
<td>$0</td>
</tr>
<tr>
<td>47</td>
<td>Restaurants</td>
<td></td>
<td></td>
<td></td>
<td>$0</td>
</tr>
<tr>
<td>48</td>
<td>Entertainment</td>
<td></td>
<td></td>
<td></td>
<td>$0</td>
</tr>
<tr>
<td>49</td>
<td>Clothes</td>
<td></td>
<td></td>
<td></td>
<td>$0</td>
</tr>
</tbody>
</table>
 ∀ Light bulb state is now pure and well encapsulated.

In a single location, we can see what it depends on (toggle) and how it changes (yellow ↔ null)

pseudo-code

```javascript
bulb.style.backgroundColor = function(toggle) {
    return toggle.value === 'on' ? 'yellow' : null;
};
```
Reactive Programming is *Dataflow Programming*

**pseudo-code**

```plaintext
a; <-- data streams in
b = a * 5;
c = b + 3;
d = a + b;
```

When new data streams in, `a` is automatically updated, and its new value propagates to variables that depend on it (`b`, `d`).

These variables update, and their values continue forwards (`b → c`).

Updates are *topologically sorted*: a variable is only updated once *all* of its dependencies have been updated.