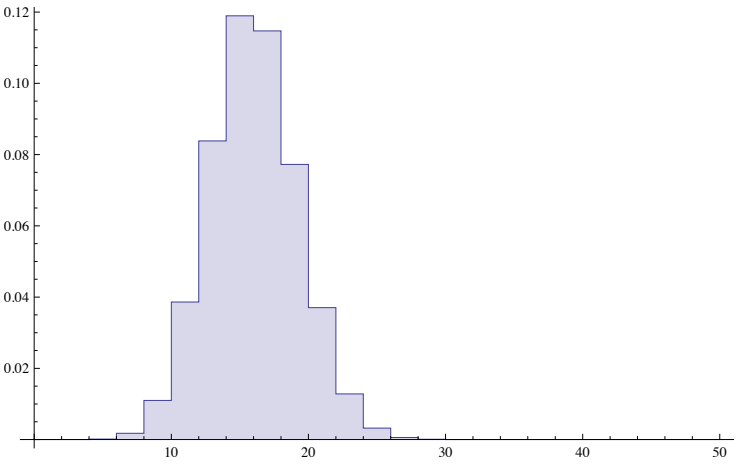
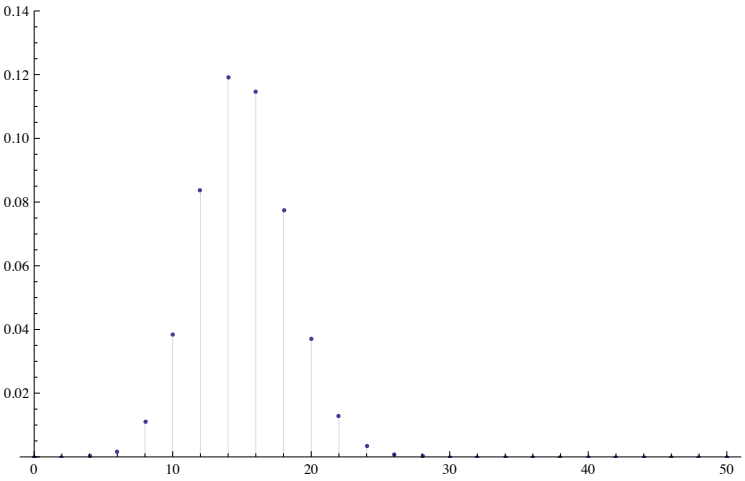
```
/*
SuperCollider Symposium- Modality Workshop
April 2012
London
© Modality Team
Creative Commons Licence Attribution-NonCommercial-ShareAlike 3.0
http://creativecommons.org/licenses/by-nc-sa/3.0/
*/


/*Function Reactive Programming*/
/*Introduction*/

FRP deals with with 'Event Processing'.
Events come in while the program is running, most often, at unpredictable times.
Under the hood it uses Observers, but it automates the removal of the observer from
the observers list.
Usually it's done with callbacks or observers. FRP is a different approach.

From Wikipedia:
Functional reactive programming (FRP) is a programming paradigm for reactive
programming using the building blocks of functional programming.

The key traits of FRP are:
   - Input is viewed as a behavior, or time-varying stream of events.
   - Continuous, time-varying values.
   - Time-ordered sequences of discrete events.
   - Time-varying values can be of higher orders. ????

'Continuous, time-varying values'

You know the value at each moment in time.
Encoded with subclasses of FPSignal.
```




```
//A Var is an FPSignal
x = Var(0.0);
//current value
x.now;
//change value
x.value_(3.3);
x.now;
```

'Time-ordered sequences of discrete events'

Encoded with subclasses of EventStream.




```
x = EventSource();
x.do{ |x| postln("I have received this: "++x) };
x.fire(1);
x.fire(5);
```

You can go from to the other with the functions 'changes' and 'hold'.

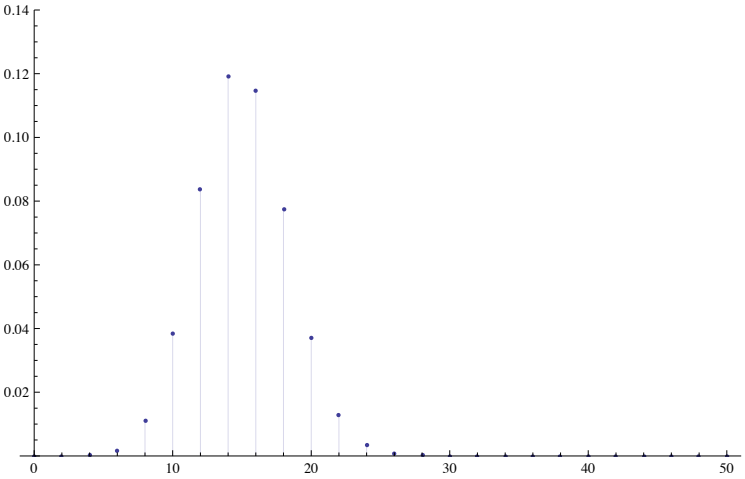
```
x = Var(0.0);
~eventStream = x.changes;

x = EventSource();
~signal = x.hold(0.0);
```

If you do

```
x = EventSource();
x.hold(0.0).changes that is equivalent to the original EventSource x.
```

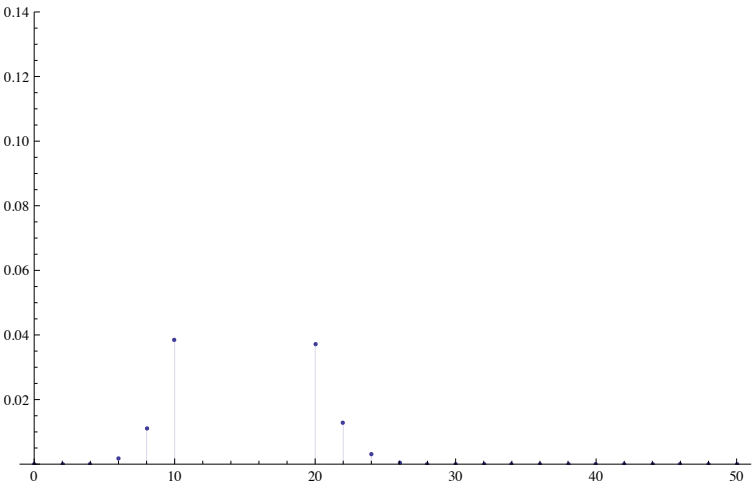
```
/*Event Streams*/

- EventStreams emit events at given times.

- EventStreams can be chained creating a network.

- The start of the network is always an EventSource.

- To react to these events you use the 'do' method.

(
x = EventSource();
x.do{ |x| postln("Hello world: "++x) };
x.fire(3)
)
```

You can think of EventStreams as Collections of elements over time. Therefore the methods of collections will work as you expect.



'select(f)' only outputs a value if f.(value) is true.

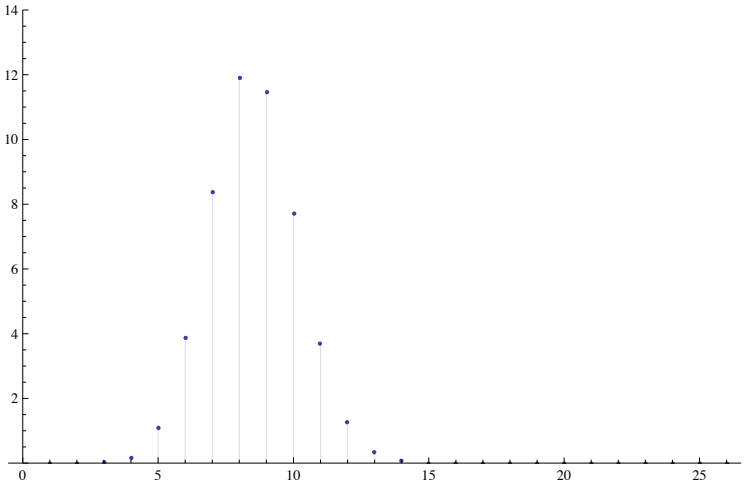
```
x.select(_<0.04)
```



```
(
x = EventSource();
y = x.select(_>3);
y.do{ |v| postln("I got a "++v) };
x.fire(1);//will not let through.
x.fire(4);//will let through.
)
```

'collect(f)' outputs a value by applying a value to the funciton f.(value).

```
x.collect(_*100)
```




```
(
x = EventSource();
y = x.collect(_*100);
y.do{ |v| postln("I got a "++v) };
x.fire(1);
x.fire(4);
)
```

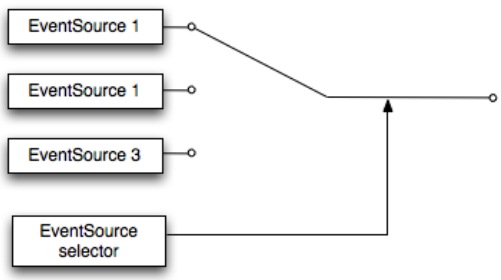
- EventSources "fire" events, which are propagated by the network of EventStreams (collect, select, etc), until they reach a do method, at which point they can finally cause side-effects. Since the network is completely functional, no visible side-effects are performed until reaching a do method.

'fold(initialState,f)' - this method can be used for keeping state. It's is the equivalent of 'inject' in sc collections.

f must receive two variables, the first is the current state and second is the last value.

```
(
x = EventSource();
y = x.fold(0.0,{ |state,v| state + v });
y.do{ |v| postln("I got a "++v) };
x.fire(1);
x.fire(4);
x.fire(7);
x.fire(12);
)

(
x = EventSource();
x.fold([],{ |state,v| state ++ [v] }).dopost;
y.do{ |v| postln("I got a "++v) };
x.fire(1);
x.fire(4);
x.fire(7);
x.fire(12);
)

//keep last two values
(
x = EventSource();
x.fold([0.0,0.0],{ |state,v| [state[1],v] }).dopost;
y.do{ |v| postln("I got a "++v) };
x.fire(1);
x.fire(4);
x.fire(7);
x.fire(12);
)
```

flatCollect(f) - This is the most important method of EventSource !! It allows selecting which events to output depending on the some other EventSource.




```
(
```

```
//selector
x = EventSource();
//two sources
y = EventSource();
z = EventSource();
//the result
w = x.flatCollect{ |v|
 if(v==0){y}{z}
};
w.do(_.postln);
)
(
//will get value from y
x.fire(0);
y.fire(1);
z.fire(4);
)
(
//will get value from z
x.fire(1);
y.fire(1);
z.fire(4);
)
```

Why is it called flatCollect ?

well, because it is analoguous to [1,2,3].collect(_*[1,2,3]).flatten, which in other languages is called flatMap or bind (It is an instance of the famous Monad). Notice that [1,2,3].collect(_*[1,2,3]) creates an array of arrays which is then flattened to just one array.
In the same way doing

```
x = EventSource();
y = EventSource();
z = EventSource();
x.collect{ |v|
 if(v==0){y}{z}
};
```

creates an EventStream of EventStreams. flatCollect will flatten it down again to a EventStream.