HWK5: Interactive Altair Plots

Name: Dyab Asdi

UT EID: da32435

Name: Raymond Sun

UT EID: rcs3634

Name: Evan Xia

UT EID: ex232

Date: 3/1/2025

The first part of this homework is to teach you the core concepts required to create a basic Altair chart; namely:

- **Data**, **Marks**, and **Encodings**: the three core pieces of an Altair chart
- **Encoding Types**: `Q` (quantitative), `N` (nominal), `O` (ordinal), `T` (temporal), which drive the visual representation of the encodings
- **Binning and Aggregation**: which let you control aspects of the data representation within Altair.

With a good understanding of these core pieces, you will be well on your way to making a variety of charts in Altair.

We will start by importing Altair:

In [1]:
```
import altair as alt
alt.__version__
```

Out[1]:
```
'5.5.0'
```

In [2]:
```
from scipy import signal
```

A Basic Altair Chart

The essential elements of an Altair chart are the **data**, the **mark**, and the **encoding**.

The format by which these are specified will look something like this:

```
alt.Chart(data).mark_point().encode(
    encoding_1='column_1',
    encoding_2='column_2',
    # etc.
)
```

Let's take a look at these pieces, one at a time.

The Data

Data in Altair is built around the Pandas Dataframe. For this section, we'll use the cars dataset, which we can load using the vega_datasets package:

In [3]:
```
from vega_datasets import data
cars = data.cars()

cars.head()
```

Out[3]:

	Name	Miles_per_Gallon	Cylinders	Displacement	Horsepower	Weight_in_lbs	Acceleration
0	chevrolet chevelle malibu	18.0	8	307.0	130.0	3504	12.0
1	buick skylark 320	15.0	8	350.0	165.0	3693	11.5
2	plymouth satellite	18.0	8	318.0	150.0	3436	11.0
3	amc rebel sst	16.0	8	304.0	150.0	3433	12.0
4	ford torino	17.0	8	302.0	140.0	3449	10.5

Data in Altair is expected to be in a tidy format; in other words:

- each **row** is an observation
- each **column** is a variable

See Altair's Data Documentation for more information.

The *Chart* object

With the data defined, you can instantiate Altair's fundamental object, the `Chart`. Fundamentally, a `Chart` is an object which knows how to emit a JSON dictionary representing the data and visualization encodings, which can be sent to the notebook and rendered by the Vega-Lite JavaScript library. Let's take a look at what this JSON representation looks like, using only the first row of the data:

In [4]:
```
cars1 = cars.iloc[:1]
alt.Chart(cars1).mark_point().to_dict()
```

Out[4]:

```
{'config': {'view': {'continuousWidth': 300, 'continuousHeight': 300}},
 'data': {'name': 'data-e88c03554d908e12891ebf77dc67f1fd'},
 'mark': {'type': 'point'},
 '$schema': 'https://vega.github.io/schema/vega-lite/v5.20.1.json',
 'datasets': {'data-e88c03554d908e12891ebf77dc67f1fd': [{'Name': 'chevrolet chevelle mali
bu',
    'Miles_per_Gallon': 18.0,
    'Cylinders': 8,
    'Displacement': 307.0,
    'Horsepower': 130.0,
    'Weight_in_lbs': 3504,
    'Acceleration': 12.0,
    'Year': '1970-01-01T00:00:00',
    'Origin': 'USA'}]}}
```


At this point the chart includes a JSON-formatted representation of the dataframe, what type of mark to use, along with some metadata that is included in every chart output.

The Mark

We can decide what sort of *mark* we would like to use to represent our data. In the previous example, we can choose the `point` mark to represent each data as a point on the plot:

In [5]:
```
alt.Chart(cars).mark_point()
```

Out[5]:


The result is a visualization with one point per row in the data, though it is not a particularly interesting: all the points are stacked right on top of each other!

It is useful to again examine the JSON output here:

In [6]:
```
alt.Chart(cars1).mark_point().to_dict()
```

Out[6]:

```
{'config': {'view': {'continuousWidth': 300, 'continuousHeight': 300}},
 'data': {'name': 'data-e88c03554d908e12891ebf77dc67f1fd'},
 'mark': {'type': 'point'},
 '$schema': 'https://vega.github.io/schema/vega-lite/v5.20.1.json',
 'datasets': {'data-e88c03554d908e12891ebf77dc67f1fd': [{'Name': 'chevrolet chevelle mali
bu',
    'Miles_per_Gallon': 18.0,
    'Cylinders': 8,
    'Displacement': 307.0,
    'Horsepower': 130.0,
    'Weight_in_lbs': 3504,
    'Acceleration': 12.0,
    'Year': '1970-01-01T00:00:00',
    'Origin': 'USA'}]}}
```


Notice that now in addition to the data, the specification includes information about the mark type.

There are a number of available marks that you can use; some of the more common are the following:

- `mark_point()`

- `mark_circle()`
- `mark_square()`
- `mark_line()`
- `mark_area()`
- `mark_bar()`
- `mark_tick()`

You can get a complete list of `mark_*` methods using Jupyter's tab-completion feature: in any cell just type:

```
alt.Chart.mark_
```

followed by the tab key to see the available options.

Encodings

The next step is to add *visual encoding channels* (or *encodings* for short) to the chart. An encoding channel specifies how a given data column should be mapped onto the visual properties of the visualization. Some of the more frequenty used visual encodings are listed here:

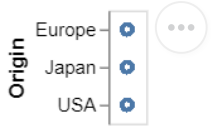
- `x` : x-axis value
- `y` : y-axis value
- `color` : color of the mark
- `opacity` : transparency/opacity of the mark
- `shape` : shape of the mark
- `size` : size of the mark
- `row` : row within a grid of facet plots
- `column` : column within a grid of facet plots

For a complete list of these encodings, see the Encodings section of the documentation.

Visual encodings can be created with the `encode()` method of the `Chart` object. For example, we can start by mapping the `y` axis of the chart to the `Origin` column:

In [7]:
```
alt.Chart(cars).mark_point().encode(
    y='Origin'
)
```

Out[7]:



The result is a one-dimensional visualization representing the values taken on by `Origin`, with the points in each category on top of each other. As above, we can view the JSON data generated for this visualization:

In [8]:
```
alt.Chart(cars1).mark_point().encode(
    x='Origin'
).to_dict()
```

Out[8]:

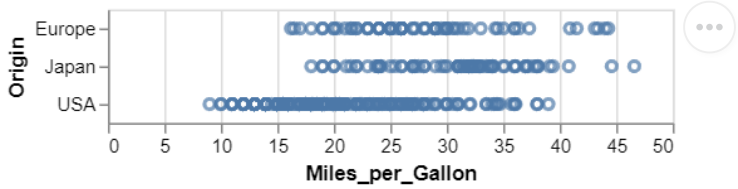
```
{'config': {'view': {'continuousWidth': 300, 'continuousHeight': 300}},
 'data': {'name': 'data-e88c03554d908e12891ebf77dc67f1fd'},
 'mark': {'type': 'point'},
 'encoding': {'x': {'field': 'Origin', 'type': 'nominal'}},
 '$schema': 'https://vega.github.io/schema/vega-lite/v5.20.1.json',
 'datasets': {'data-e88c03554d908e12891ebf77dc67f1fd': [{'Name': 'chevrolet chevelle mali
bu',
    'Miles_per_Gallon': 18.0,
    'Cylinders': 8,
    'Displacement': 307.0,
    'Horsepower': 130.0,
    'Weight_in_lbs': 3504,
    'Acceleration': 12.0,
    'Year': '1970-01-01T00:00:00',
    'Origin': 'USA'}]}}
```


The result is the same as above with the addition of the `'encoding'` key, which specifies the visualization channel (`y`), the name of the field (`Origin`), and the type of the variable (`nominal`). We'll discuss these data types in a moment.

The visualization can be made more interesting by adding another channel to the encoding: let's encode the `Miles_per_Gallon` as the `x` position:

In [9]:
```
alt.Chart(cars).mark_point().encode(
    y='Origin',
    x='Miles_per_Gallon'
)
```

Out[9]:



You can add as many encodings as you wish, with each encoding mapped to a column in the data. For example, here we will color the points by *Origin*, and plot *Miles_per_gallon* vs *Year*:

In [10]:
```
alt.Chart(cars).mark_point().encode(
    color='Origin',
    y='Miles_per_Gallon',
    x='Year'
)
```

Out[10]:

Encoding Types

One of the central ideas of Altair is that the library will **choose good defaults for your data type**.

The basic data types supported by Altair are as follows:

Data Type	Code	Description
quantitative	Q	Numerical quantity (real-valued)
nominal	N	Name / Unordered categorical
ordinal	O	Ordered categorial
temporal	T	Date/time

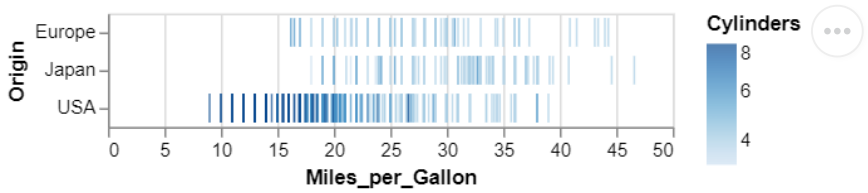
When you specify data as a pandas dataframe, these types are **automatically determined** by Altair.

When you specify data as a URL, you must **manually specify** data types for each of your columns.

Let's look at a simple plot containing three of the columns from the cars data:

In [11]:
```
alt.Chart(cars).mark_tick().encode(
    x='Miles_per_Gallon',
    y='Origin',
    color='Cylinders'
)
```

Out[11]:



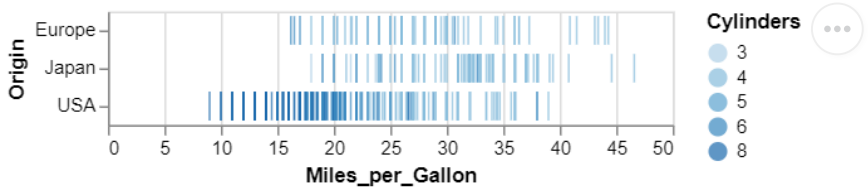
Questions to ponder:

- what data type best goes with `Miles_per_Gallon`?
- what data type best goes with `Origin`?
- what data type best goes with `Cylinders`?

Let's add the shorthands for each of these data types to our specification, using the one-letter codes above (for example, change `"Miles_per_Gallon"` to `"Miles_per_Gallon:Q"` to explicitly specify that it is a quantitative type):

In [12]:
```
alt.Chart(cars).mark_tick().encode(
    x='Miles_per_Gallon:Q',
    y='Origin:N',
    color='Cylinders:O'
)
```

Out[12]:



Notice how if we change the data type for `'Cylinders'` to ordinal the plot changes.

As you use Altair, it is useful to get into the habit of always specifying these types explicitly, because this is *mandatory* when working with data loaded from a file or a URL.

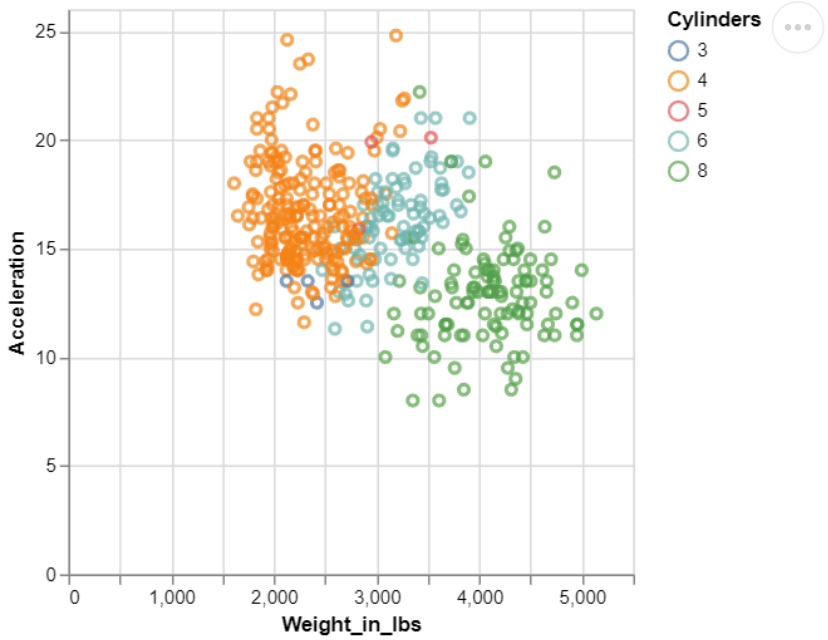
Exercises

Create the following Graphs:

1. A Scatter plot of "Weight_in_lbs" on the x-axis, and "Acceleration" in the y-axis with the number of cylinders encoded in the color channel of all the cars in the dataframe. **(10 marks)**
2. A horizontal histogram of the country of origin (this documentation might help) of all the cars in the dataframe. **(15 marks)**
3. A Heatmap of the binned horsepower (x) and the binned displacement (y) with the count representing color. (this documentation might help). Use 20 bins for each dimension. **(15 marks)**

In [56]:
```
# Code for visualization 1
alt.Chart(cars).mark_point().encode(
    x= 'Weight_in_lbs:Q',
    y= 'Acceleration:Q',
    color='Cylinders:N'
)
```

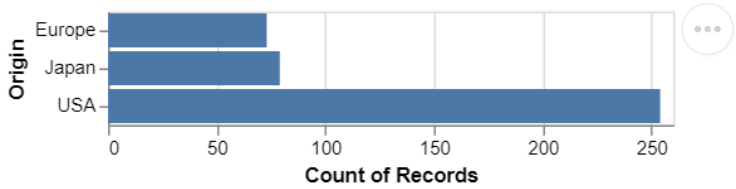
Out[56]:



In [58]:

```
# Code for visualization 2
alt.Chart(cars).mark_bar().encode(
    x= 'count()',
    y='Origin:N',
)
```

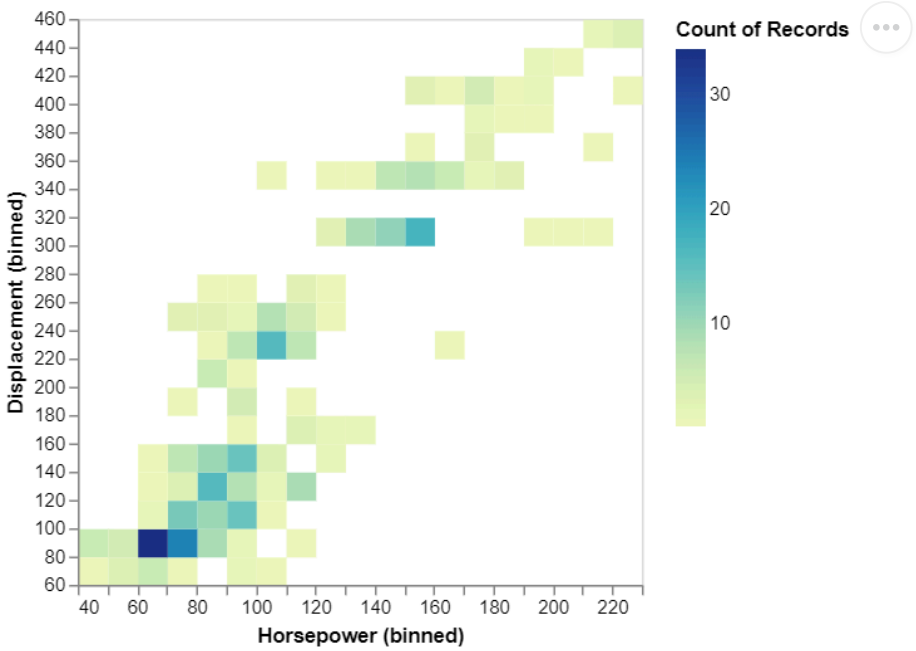
Out[58]:



In [62]:

```
# Code for visualization 3
alt.Chart(cars).mark_rect().encode(
    x=alt.X('Horsepower:Q', bin=alt.Bin(maxbins=20)),
    y=alt.Y('Displacement:Q', bin=alt.Bin(maxbins=20)),
    color='count()'
)
```

Out[62]:



Compound Charts

Altair provides a concise API for creating multi-panel and layered charts, and we'll mention three of them explicitly here:

- Layering
- Horizontal Concatenation
- Vertical Concatenation
- Repeat Charts

We'll explore those briefly here.

Layering

Layering lets you put layer multiple marks on a single Chart. One common example is creating a plot with both points and lines representing the same data.

Let's use the `stocks` data for this example:

In [16]:
```
from vega_datasets import data
stocks = data.stocks()
stocks.head()
```

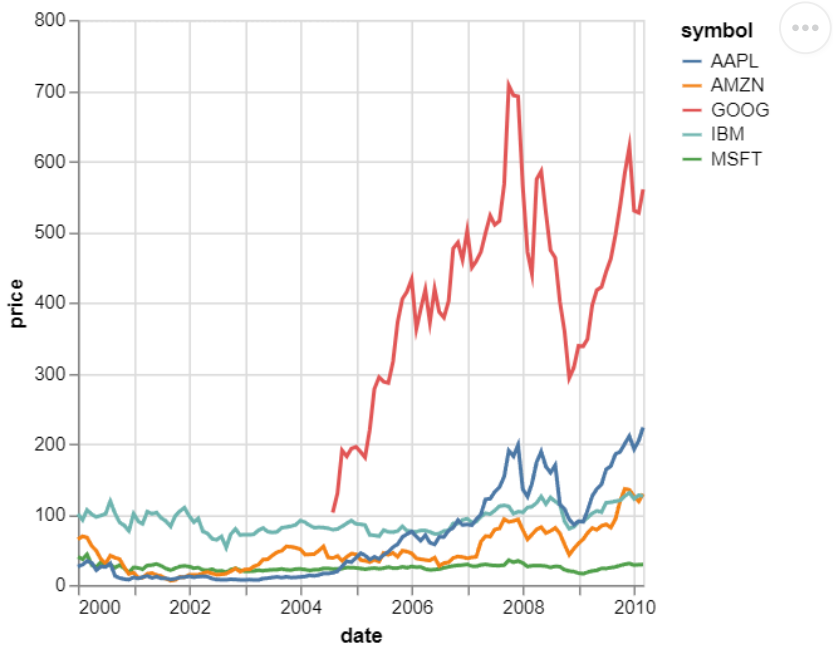
Out[16]:

	symbol	date	price
0	MSFT	2000-01-01	39.81
1	MSFT	2000-02-01	36.35
2	MSFT	2000-03-01	43.22
3	MSFT	2000-04-01	28.37
4	MSFT	2000-05-01	25.45

Here is a simple line plot for the stocks data:

In [17]:
```
alt.Chart(stocks).mark_line().encode(
    x='date:T',
    y='price:Q',
    color='symbol:N'
)
```

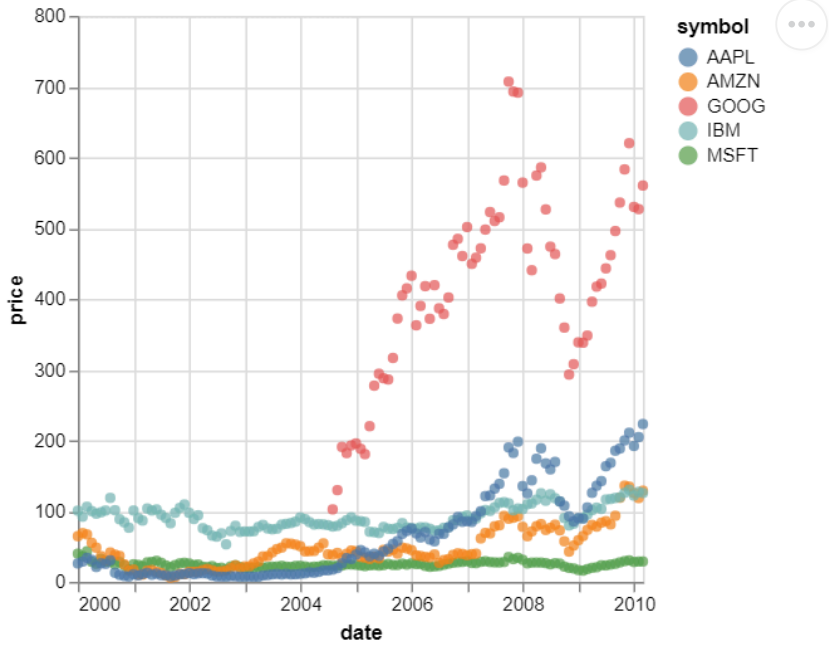
Out[17]:



and here is the same plot with a `circle` mark:

In [18]:
```
alt.Chart(stocks).mark_circle().encode(
    x='date:T',
    y='price:Q',
    color='symbol:N'
)
```

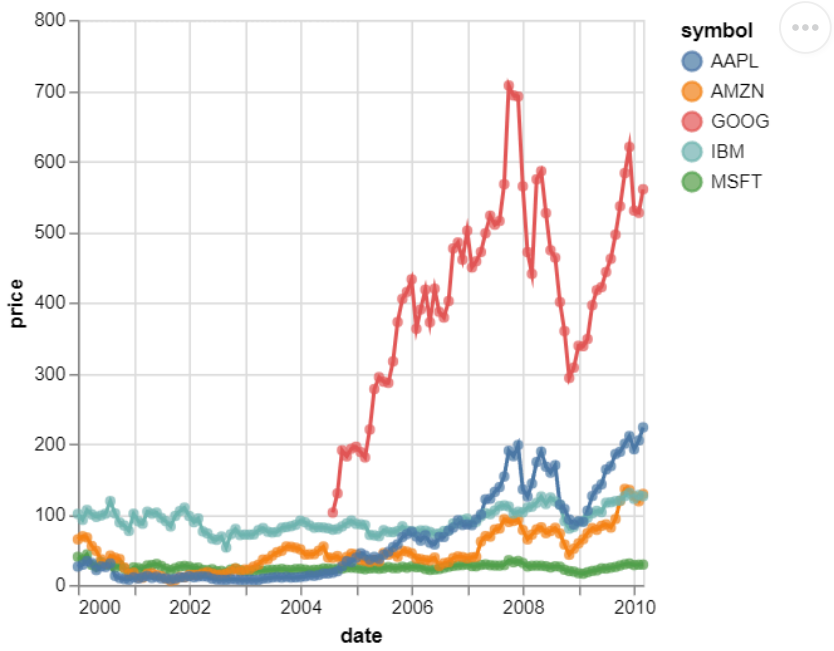
Out[18]:



We can layer these two plots together using a `+` operator:

In [19]:
```
lines = alt.Chart(stocks).mark_line().encode(
    x='date:T',
    y='price:Q',
    color='symbol:N'
)

points = alt.Chart(stocks).mark_circle().encode(
    x='date:T',
    y='price:Q',
    color='symbol:N'
)

lines + points
```

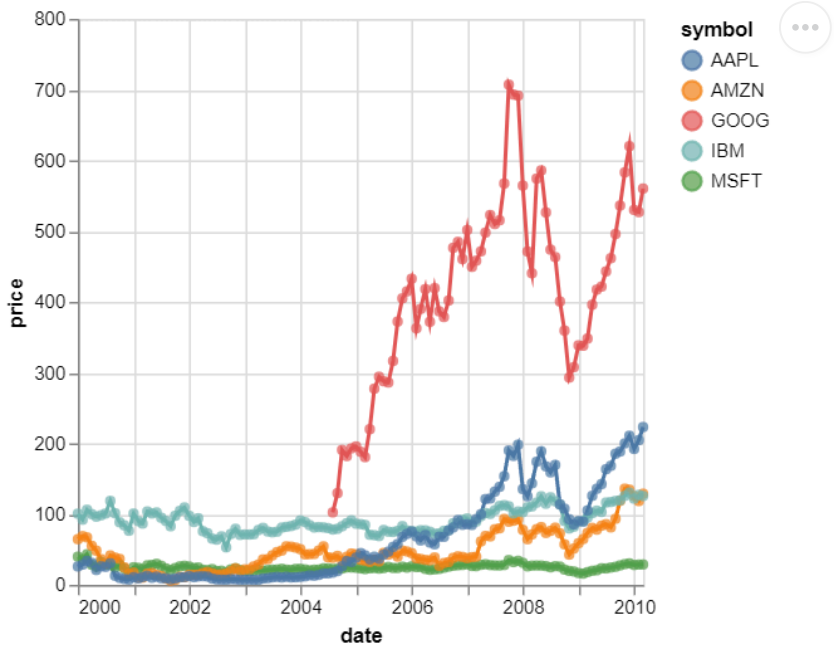
Out[19]:



This + is just a shortcut to the alt.layer() function, which does the same thing:

In [20]:
```
alt.layer(lines, points)
```

Out[20]:

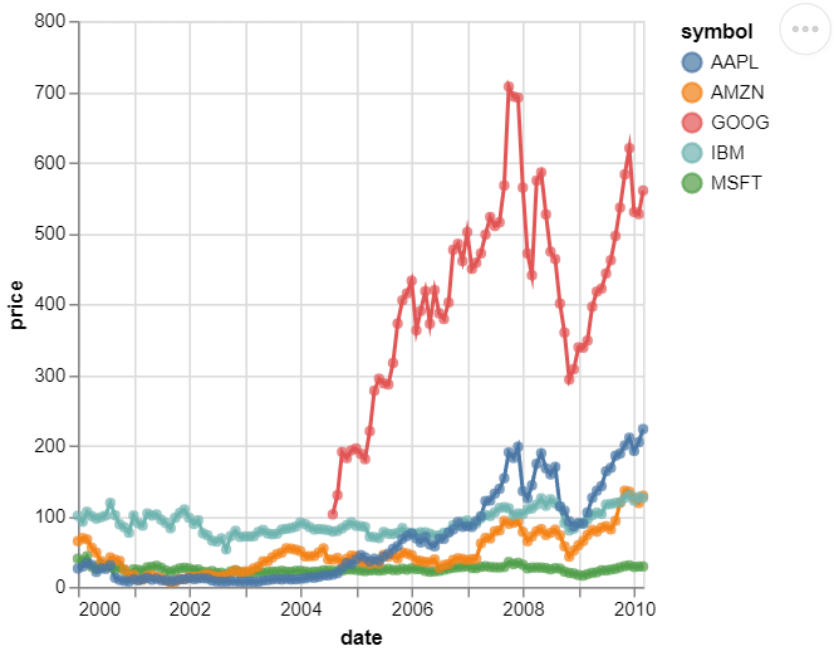


One pattern we will use often is to create a base chart with the common elements, and add together two copies with just a single change:

In [21]:

```
base = alt.Chart(stocks).encode(
    x='date:T',
    y='price:Q',
    color='symbol:N'
)

base.mark_line() + base.mark_circle()
```

Out[21]:

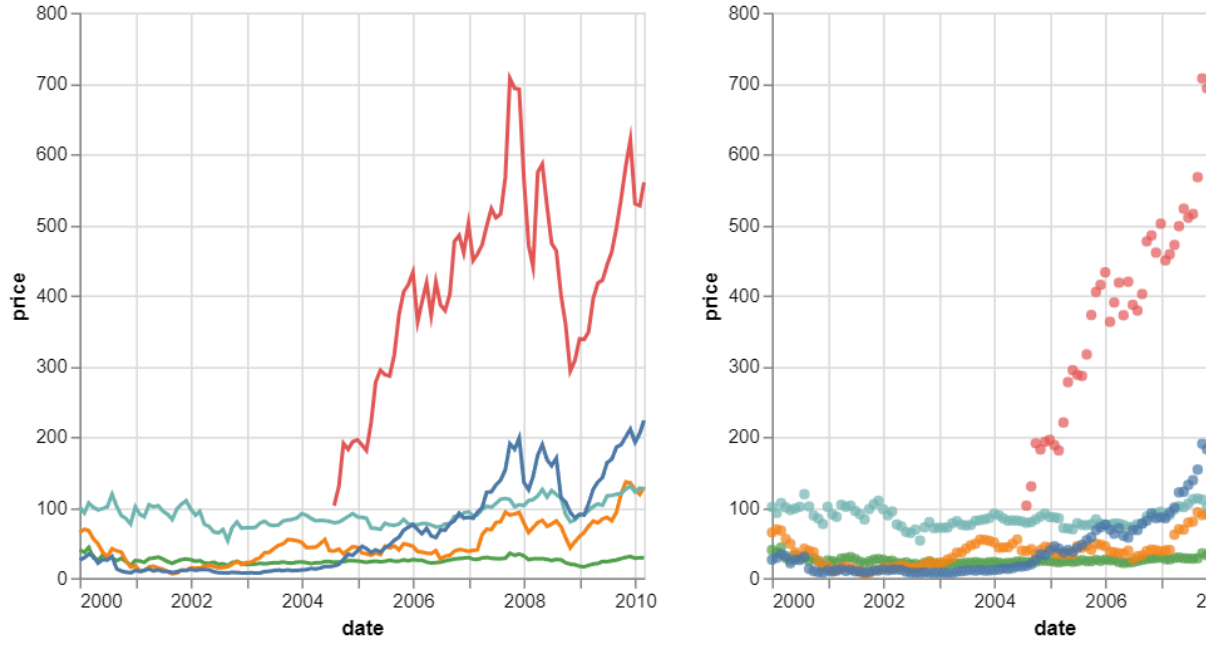


Horizontal Concatenation

Just as we can layer charts on top of each other, we can concatenate horizontally using `alt.hconcat`, or equivalently the `|` operator:

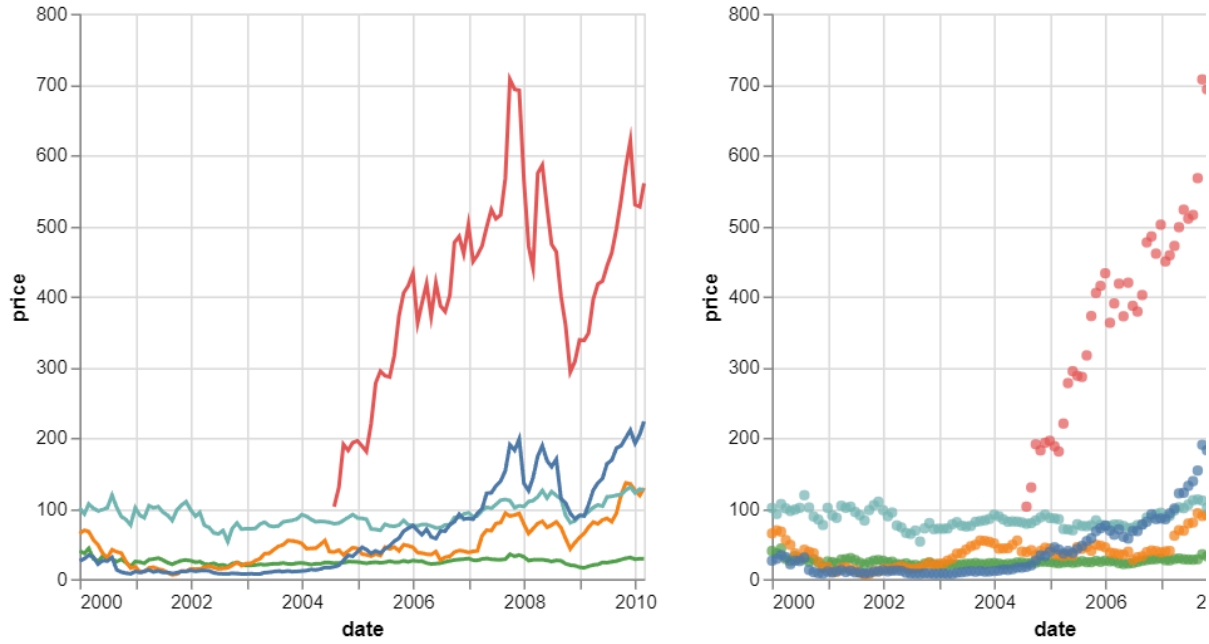
In [22]:
```
base.mark_line() | base.mark_circle()
```

Out[22]:



In [23]:
```
alt.hconcat(base.mark_line(),
            base.mark_circle())
```

Out[23]:



This can be most useful for creating multi-panel views; for example, here is the iris dataset:

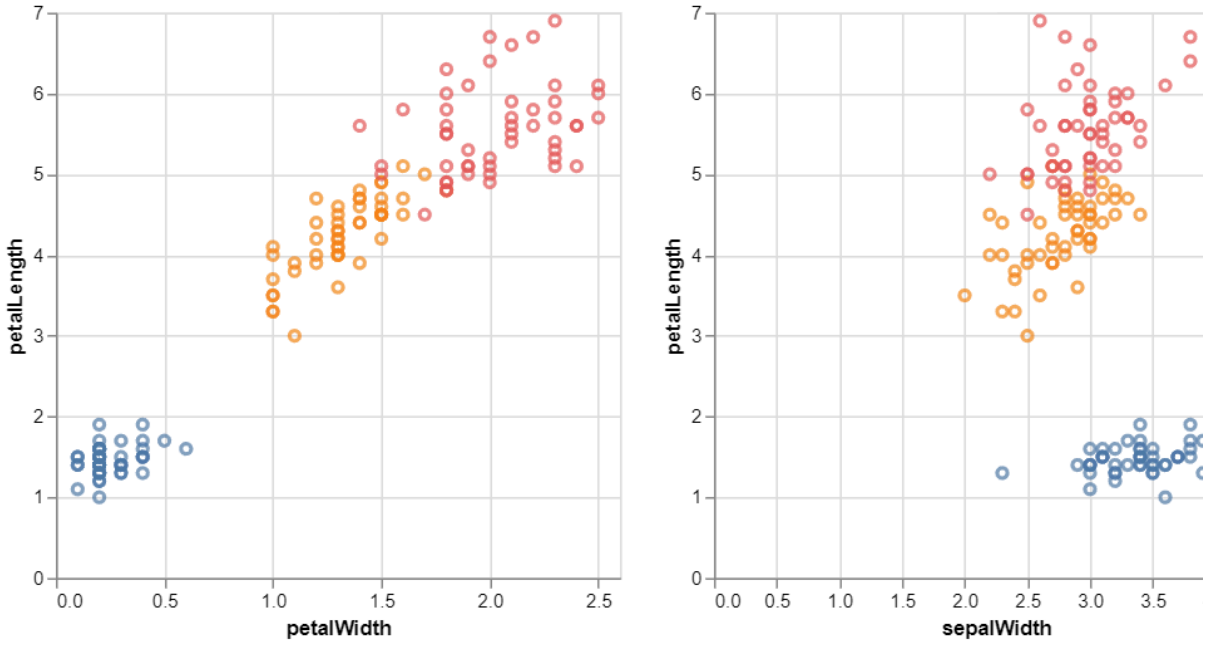
In [24]:
```
iris = data.iris()
iris.head()
```

Out[24]:

	sepalLength	sepalWidth	petalLength	petalWidth	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

In [25]:
```
base = alt.Chart(iris).mark_point().encode(
    x='petalWidth',
    y='petalLength',
    color='species'
)

base | base.encode(x='sepalWidth')
```

Out[25]:

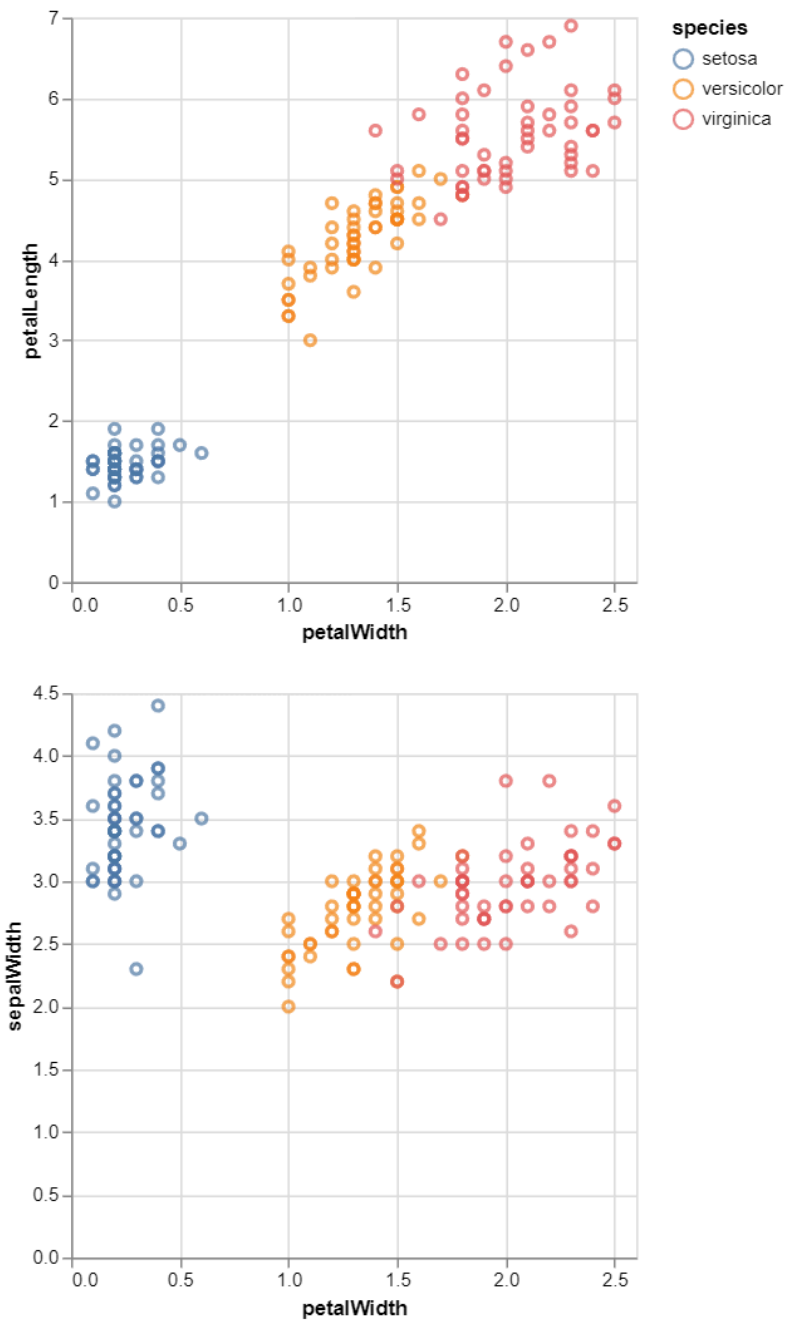


Vertical Concatenation

Vertical concatenation looks a lot like horizontal concatenation, but using either the `alt.hconcat()` function, or the `&` operator:

In [26]:
```
base & base.encode(y='sepalWidth')
```

Out[26]:



Repeat Chart - Small Multiples (SPLOM)

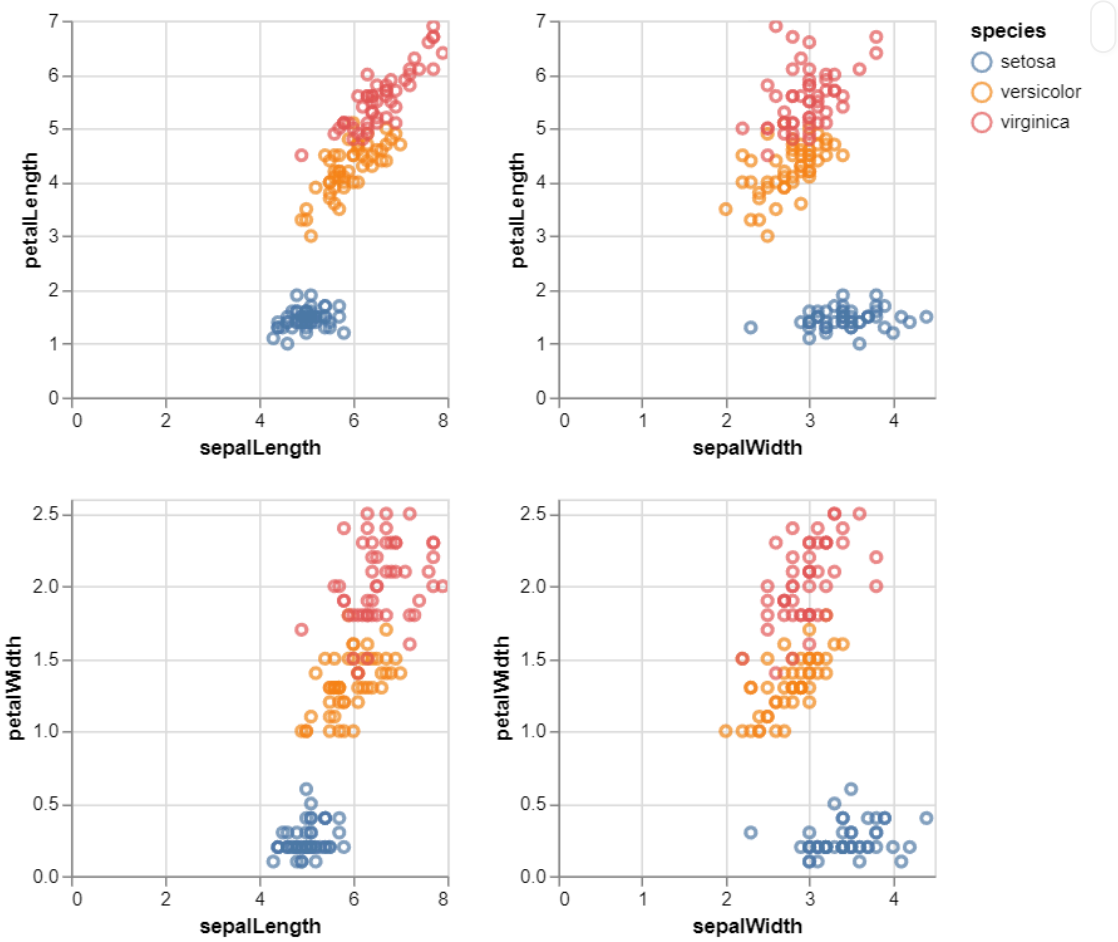
Because it is such a common pattern to horizontally and vertically concatenate charts while changing one encoding, Altair offers a shortcut for this, using the `repeat()` operator.

In [27]:

```
import altair as alt
from vega_datasets import data

iris = data.iris()

fields = ['petalLength', 'petalWidth', 'sepalLength', 'sepalWidth']

alt.Chart(iris).mark_point().encode(
    alt.X(alt.repeat("column"), type='quantitative'),
    alt.Y(alt.repeat("row"), type='quantitative'),
    color='species:N'
```

```
).properties(
    width=200,
    height=200
).repeat(
    row=['petalLength', 'petalWidth'],
    column=['sepalLength', 'sepalWidth']
).interactive()
```

Out[27]:



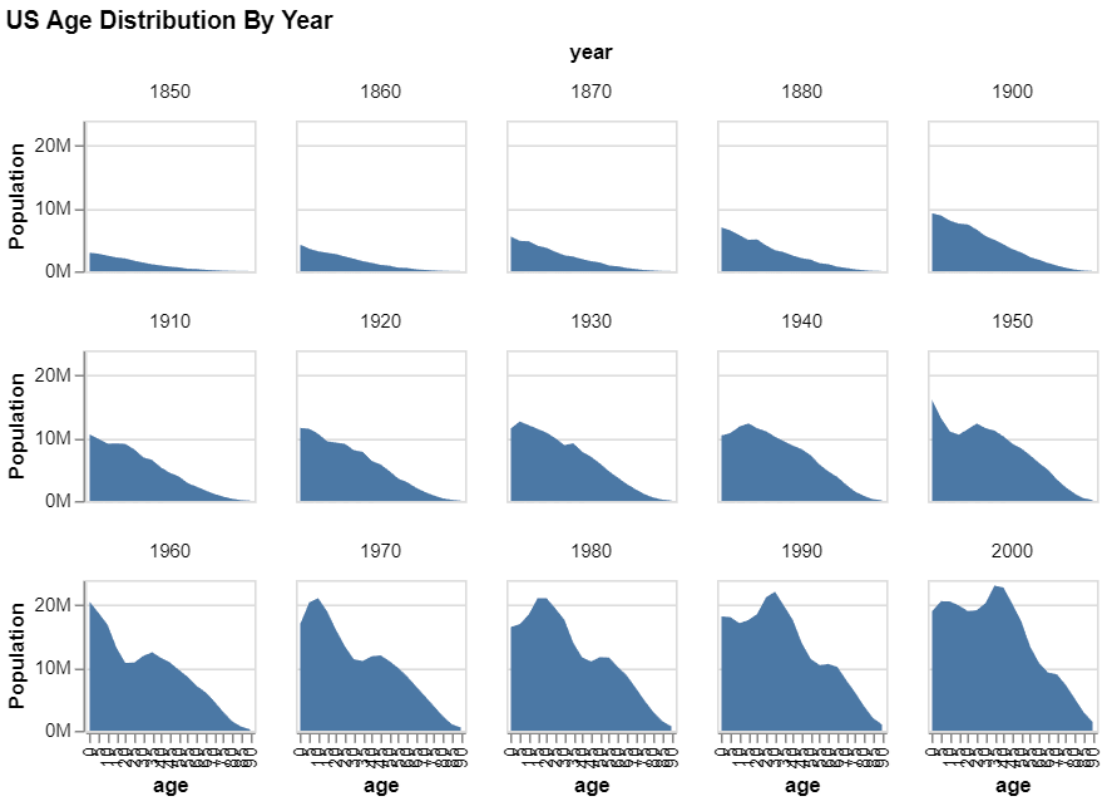
Small Multiples - US Population: Wrapped Facet

This chart visualizes the age distribution of the US population over time, using a wrapped faceting of the data by decade.

In [28]:

```
population = data.population.url

alt.Chart(population).mark_area().encode(
    x='age:O',
    y=alt.Y(
        'sum(people):Q',
        title='Population',
        axis=alt.Axis(format='~s')
    ),
    facet=alt.Facet('year:O', columns=5),
).properties(
    title='US Age Distribution By Year',
    width=90,
    height=80
)
```

Out[28]:



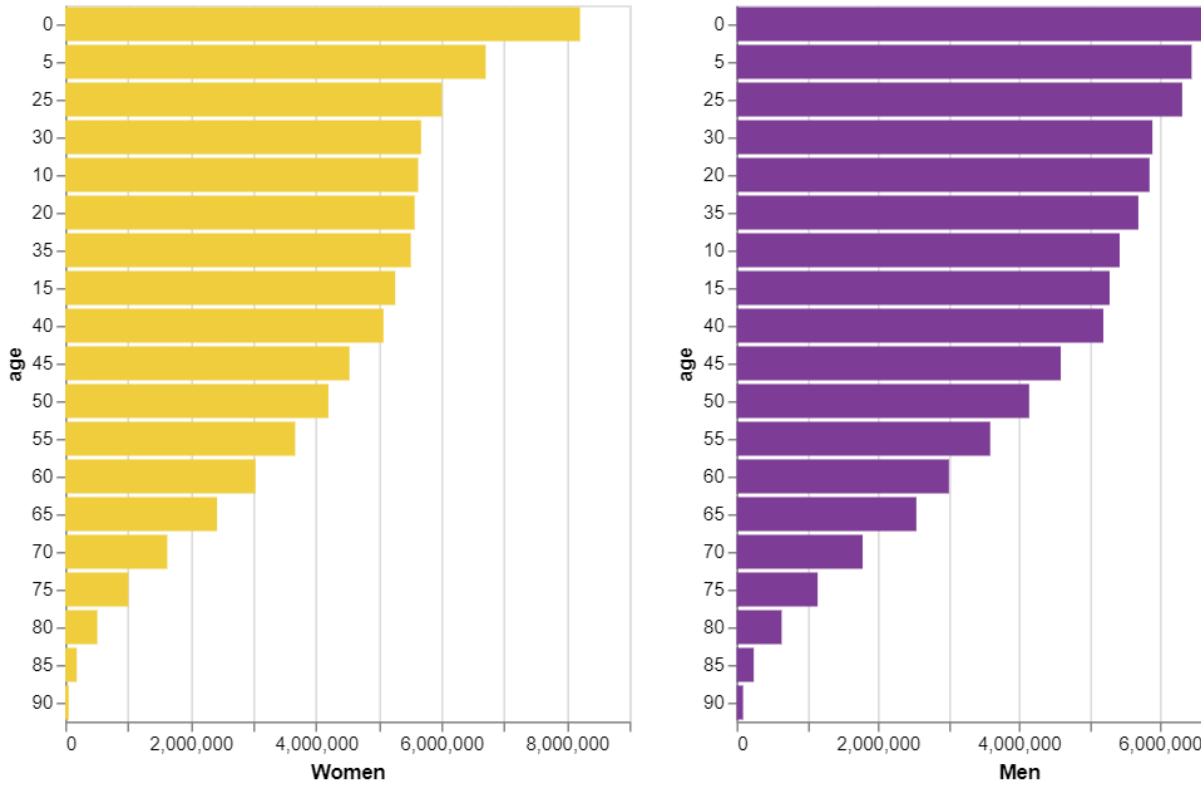
Exercise

4. Create horizontally juxtaposed bar charts using the population data. The left bar chart should show the age and number of people in 1950 that are women (`sex` == 1) as a function of age, and the right bar chart should show the number of people in 1950 that are men (`sex` == 2) as a function of age. Make the left bar chart be the color '#F4D03F' and the right bar chart to be the color '#7D3C98'. Filter the data using the slicing techniques in pandas. **(15 marks)**
5. Create a layered line graph that shows the number of women in 1950 as a function of age and the number of men in 1950 as a function of age. Use the same colors as the first problem. It's ok for now if you don't have a legend (legends in layered altair charts are complicated). Again, use the pandas slicing to filter the data prior to creating the chart. **(15 marks)**

In [74]:

```
# your code here for horizontally juxtaposed bar charts
population_df = data.population()
# filter data to get 1950 only data, for men and women
population_1950 = population_df[population_df['year'] == 1950]
women_data = population_1950[population_1950['sex'] == 1]
men_data = population_1950[population_1950['sex'] == 2]


women_graph = alt.Chart(women_data).mark_bar(color='#F4D03F').encode(
    x= alt.X('people:Q', title='Women'),
    y= alt.Y('age:O', sort= '-x'),
)
men_graph = alt.Chart(men_data).mark_bar(color='#7D3C98').encode(
    x= alt.X('people:Q', title='Men'),
    y= alt.Y('age:O', sort= '-x'),
)

alt.hconcat(women_graph, men_graph)
```

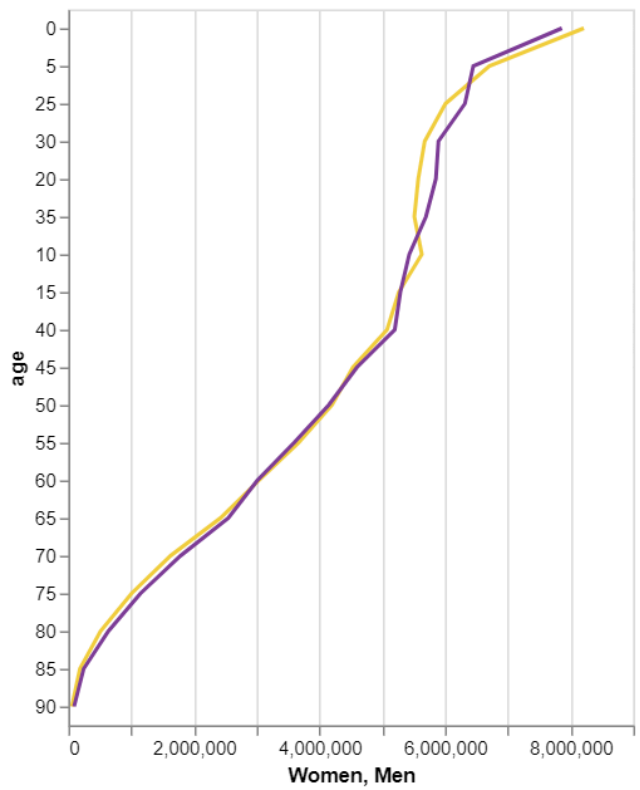
Out[74]:



In [76]:

```
# your code here for layered line graph
women_line = alt.Chart(women_data).mark_line(color='#F4D03F').encode(
    x= alt.X('people:Q', title='Women'),
    y= alt.Y('age:O', sort= '-x'),
)
men_line = alt.Chart(men_data).mark_line(color='#7D3C98').encode(
    x= alt.X('people:Q', title='Men'),
    y= alt.Y('age:O', sort= '-x'),
)

combined = women_line + men_line
combined
```

Out[76]:



Interactivity and Selections

Altair's interactivity and grammar of selections are one of its unique features among available plotting libraries. In this section, we will walk through the variety of selection types that are available, and begin to practice creating interactive charts and dashboards.

There are three basic types of selections available:

- Interval Selection: `alt.selection_interval()`
- Single Selection: `alt.selection_single()`
- Multi Selection: `alt.selection_multi()`

And we will cover four basic things that you can do with these selections

- Conditional encodings
- Scales
- Filters
- Domains

In [31]:
```
import altair as alt
from vega_datasets import data
```

Basic Interactions: Panning, Zooming, Tooltips

The basic interactions that Altair makes available are panning, zooming, and tooltips. This can be done in your chart without any use of the selection interface, using the `interactive()` shortcut method and the `tooltip` encoding.

For example, with our standard cars dataset, we can do the following:

In [32]:
```
cars = data.cars()
cars.head()
```

Out[32]:

	Name	Miles_per_Gallon	Cylinders	Displacement	Horsepower	Weight_in_lbs	Acceleration
0	chevrolet chevelle malibu	18.0	8	307.0	130.0	3504	12.0
1	buick skylark 320	15.0	8	350.0	165.0	3693	11.5
2	plymouth satellite	18.0	8	318.0	150.0	3436	11.0
3	amc rebel sst	16.0	8	304.0	150.0	3433	12.0
4	ford torino	17.0	8	302.0	140.0	3449	10.5

In [33]:
```
alt.Chart(cars).mark_point().encode(
    x='Horsepower:Q',
    y='Miles_per_Gallon:Q',
    color='Origin',
    tooltip='Name'
).interactive()
```

Out[33]:

At this point, hovering over a point will bring up a tooltip with the name of the car model, and clicking/dragging/scrolling will pan and zoom on the plot.

More Sophisticated Interaction: Selections

Basic Selection Example: Interval

As an example of a selection, let's add an interval selection to a chart.

We'll start with our cannonical scatter plot:

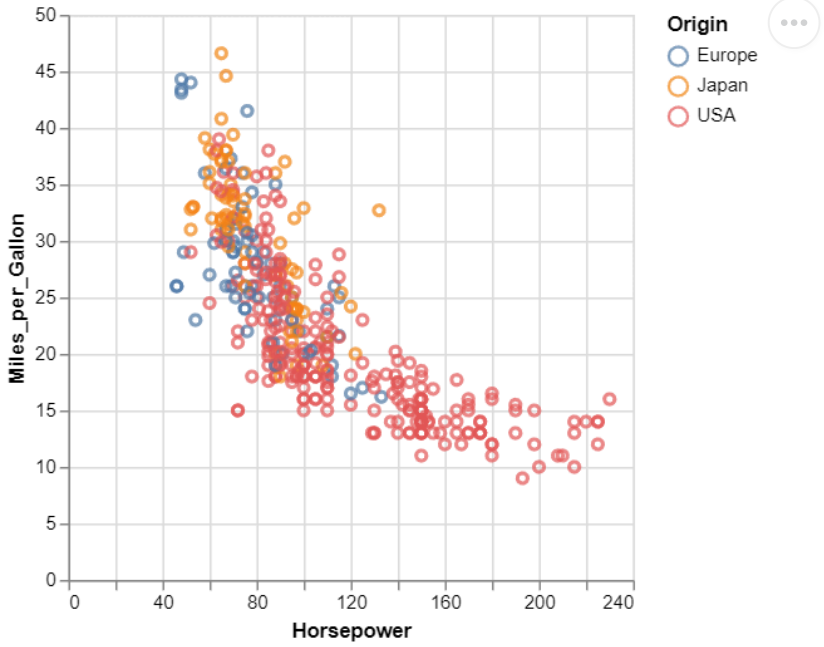
In [34]:
```
cars = data.cars()
cars.head()
```

Out[34]:

	Name	Miles_per_Gallon	Cylinders	Displacement	Horsepower	Weight_in_lbs	Acceleration
0	chevrolet chevelle malibu	18.0	8	307.0	130.0	3504	12.0
1	buick skylark 320	15.0	8	350.0	165.0	3693	11.5
2	plymouth satellite	18.0	8	318.0	150.0	3436	11.0
3	amc rebel sst	16.0	8	304.0	150.0	3433	12.0
4	ford torino	17.0	8	302.0	140.0	3449	10.5

In [35]:
```
alt.Chart(cars).mark_point().encode(
    x='Horsepower:Q',
    y='Miles_per_Gallon:Q',
    color='Origin'
)
```

Out[35]:



To add selection behavior to a chart, we create the selection object and use the `add_selection` method:

In [36]:
```
interval = alt.selection_interval()

alt.Chart(cars).mark_point().encode(
    x='Horsepower:Q',
    y='Miles_per_Gallon:Q',
    color='Origin'
).add_params(
    interval
)
```

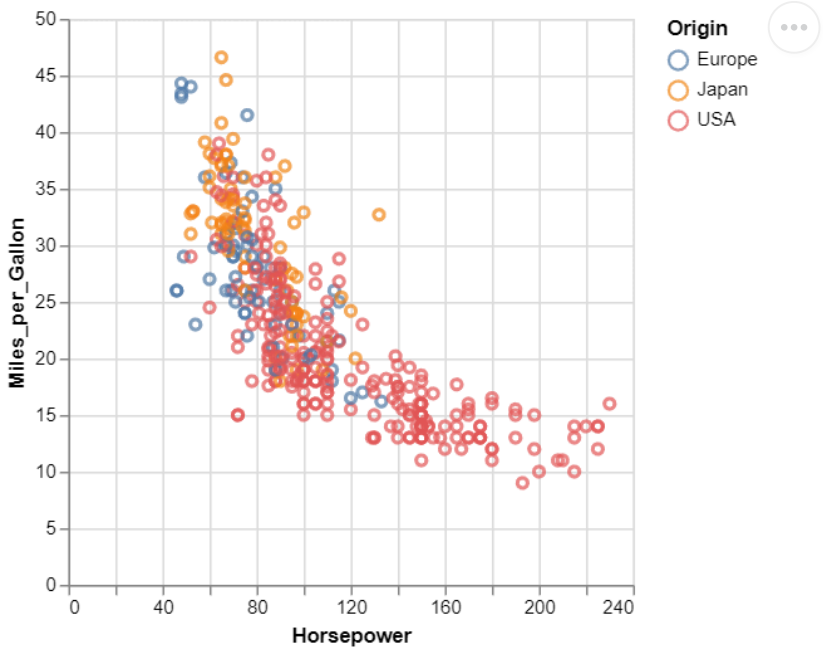
Out[36]:

This adds an interaction to the plot that lets us select points on the plot; perhaps the most common use of a selection is to highlight points by conditioning their color on the result of the selection.

This can be done with `alt.condition` :

In [37]:
```
interval = alt.selection_interval()

alt.Chart(cars).mark_point().encode(
    x='Horsepower:Q',
    y='Miles_per_Gallon:Q',
    color=alt.condition(interval, 'Origin', alt.value('lightgray'))
).add_params (
    interval
)
```

Out[37]:



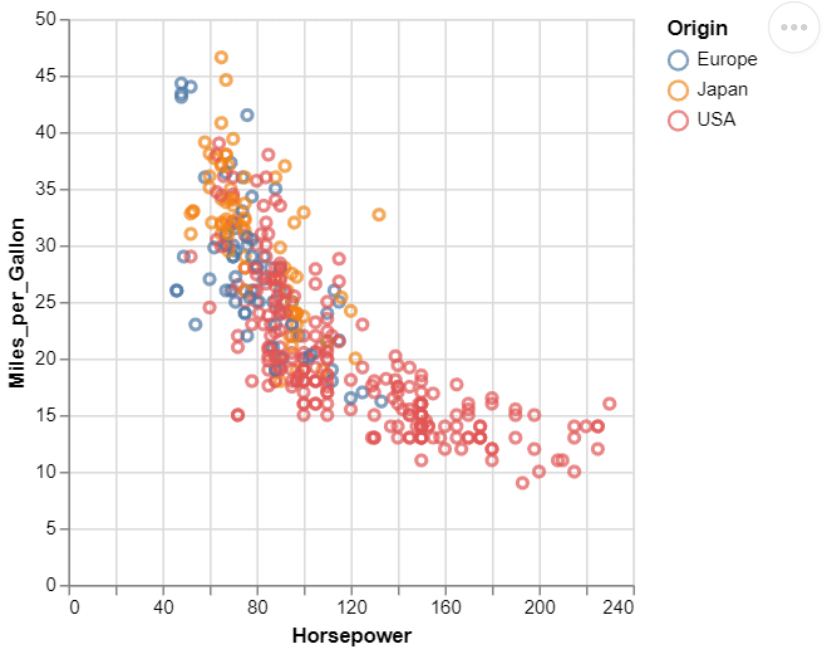
The `alt.condition` function takes three arguments: a selection object, a value to be applied to points within the selection, and a value to be applied to points outside the selection. Here we use `alt.value('lightgray')` to make certain that the color is treated as an actual color, rather than the name of a data column.

Customizing the Interval selection

The `alt.selection_interval()` function takes a number of additional arguments; for example, by specifying `encodings`, we can control whether the selection covers x, y, or both:

In [38]:
```
interval = alt.selection_interval(encodings=['x'])

alt.Chart(cars).mark_point().encode(
    x='Horsepower:Q',
    y='Miles_per_Gallon:Q',
    color=alt.condition(interval, 'Origin', alt.value('lightgray'))
).add_params(
    interval
)
```

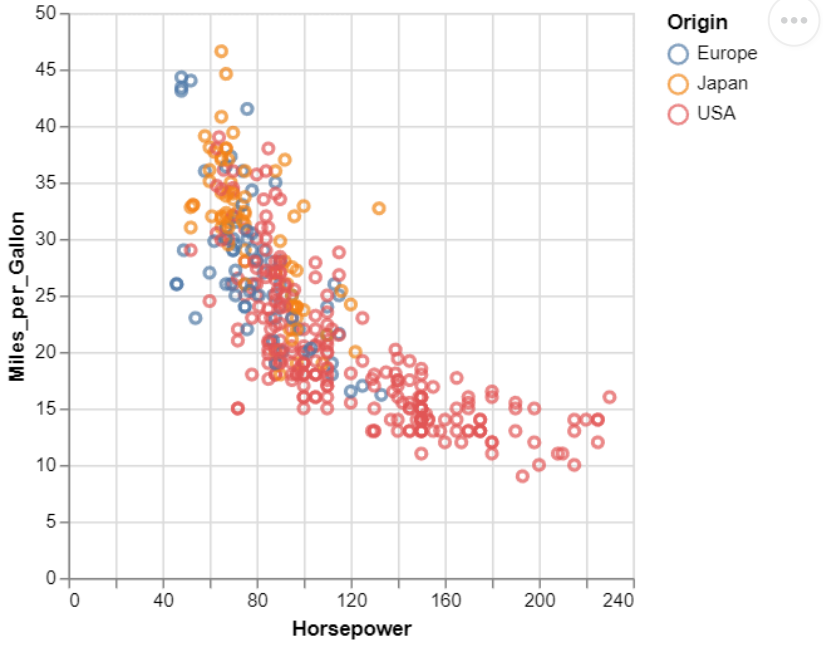
Out[38]:



In [39]:

```
interval = alt.selection_interval(encodings=['y'])

alt.Chart(cars).mark_point().encode(
    x='Horsepower:Q',
    y='Miles_per_Gallon:Q',
    color=alt.condition(interval, 'Origin', alt.value('lightgray'))
).add_params(
    interval
)
```

Out[39]:



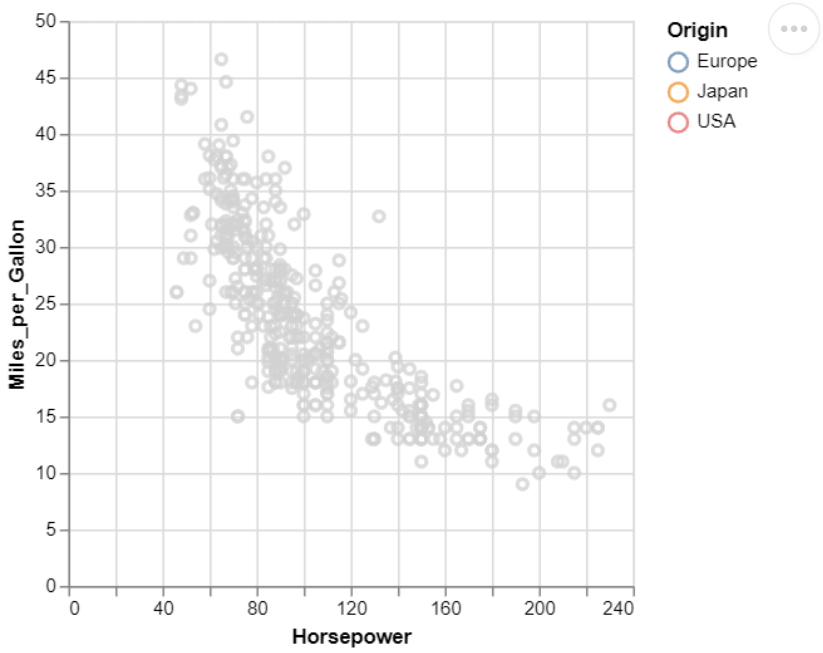
The `empty` argument lets us control whether empty selections contain *all* values, or *none* of the values; with `empty='none'` points are grayed-out by default:

In [40]:

```
interval = alt.selection_interval(empty = False)

alt.Chart(cars).mark_point().encode(
```

```
    x='Horsepower:Q',
    y='Miles_per_Gallon:Q',
    color=alt.condition(interval, 'Origin', alt.value('lightgray'))
).add_params(
    interval
)
```

Out[40]:



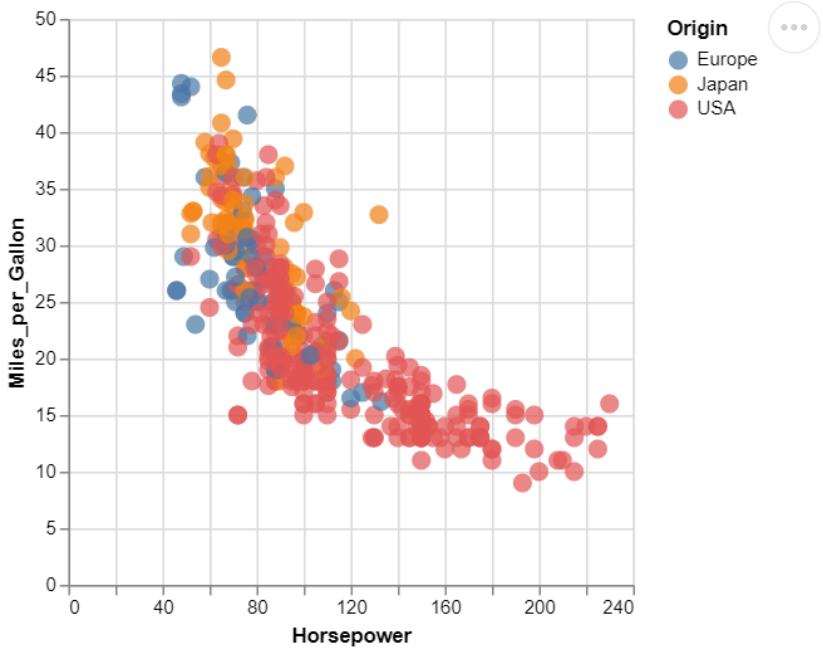
Single Selections

The `alt.selection_point()` function allows the user to click on single chart objects to select them, one at a time. We'll make the points a bit bigger so they are easier to click:

In [41]:

```
single = alt.selection_point()

alt.Chart(cars).mark_circle(size=100).encode(
    x='Horsepower:Q',
    y='Miles_per_Gallon:Q',
    color=alt.condition(single, 'Origin', alt.value('lightgray'))
).add_params(
    single
)
```

Out[41]:

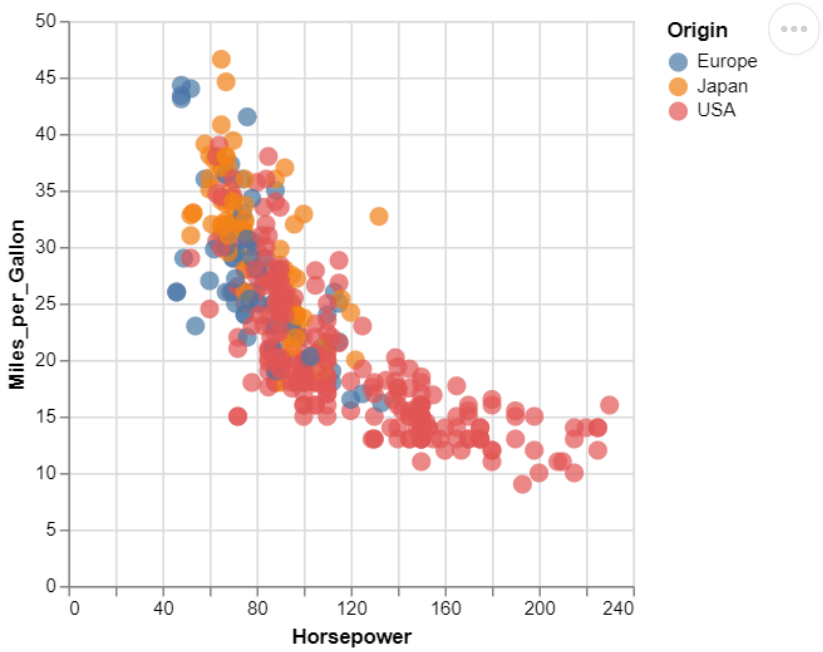


The single selection allows other behavior as well; for example, we can set `nearest=True` and `on='mouseover'` to update the highlight to the nearest point as we move the mouse:

In [42]:

```
single = alt.selection_point(on='mouseover', nearest=True)

alt.Chart(cars).mark_circle(size=100).encode(
    x='Horsepower:Q',
    y='Miles_per_Gallon:Q',
    color=alt.condition(single, 'Origin', alt.value('lightgray'))
).add_params(
    single
)
```

Out[42]:

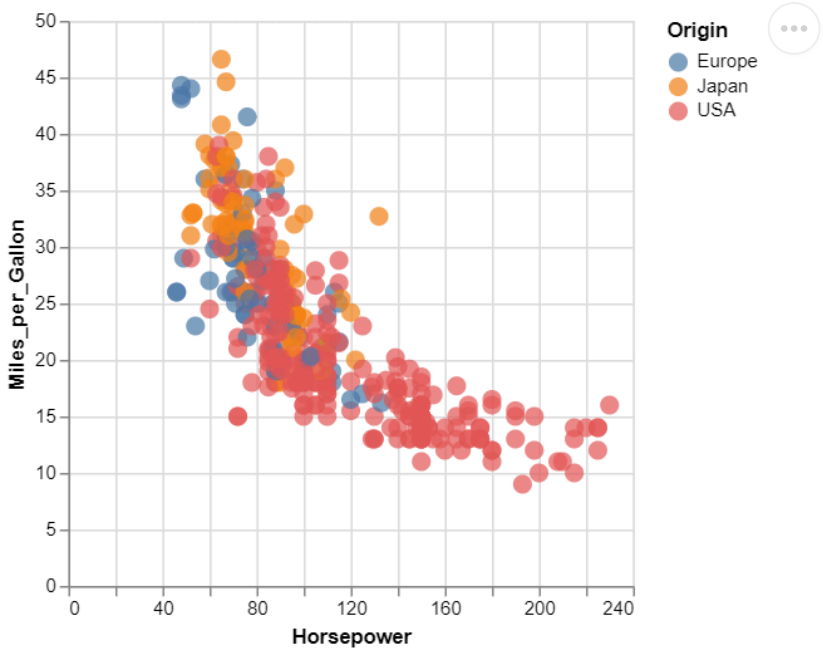


Multi Selection

Use `alt.selection_point()` function to select multiple points at once, while holding the shift key:

In [43]:
```
multi = alt.selection_point()

alt.Chart(cars).mark_circle(size=100).encode(
    x='Horsepower:Q',
    y='Miles_per_Gallon:Q',
    color=alt.condition(multi, 'Origin', alt.value('lightgray'))
).add_params(
    multi
)
```

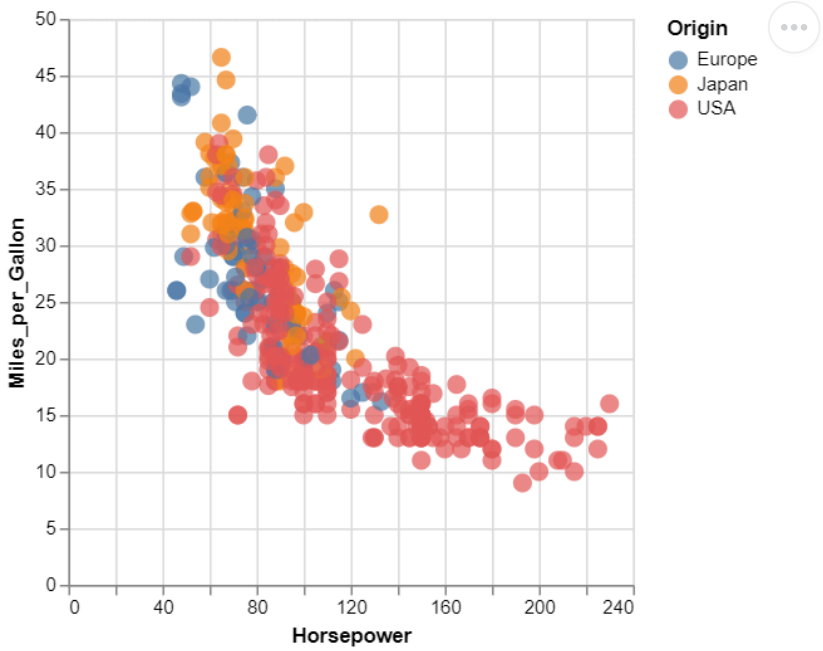
Out[43]:



Options like `on` and `nearest` also work for multi selections:

In [44]:
```
multi = alt.selection_point(on='mouseover', nearest=True)

alt.Chart(cars).mark_circle(size=100).encode(
    x='Horsepower:Q',
    y='Miles_per_Gallon:Q',
    color=alt.condition(multi, 'Origin', alt.value('lightgray'))
).add_params(
    multi
)
```

Out[44]:



Selection Binding

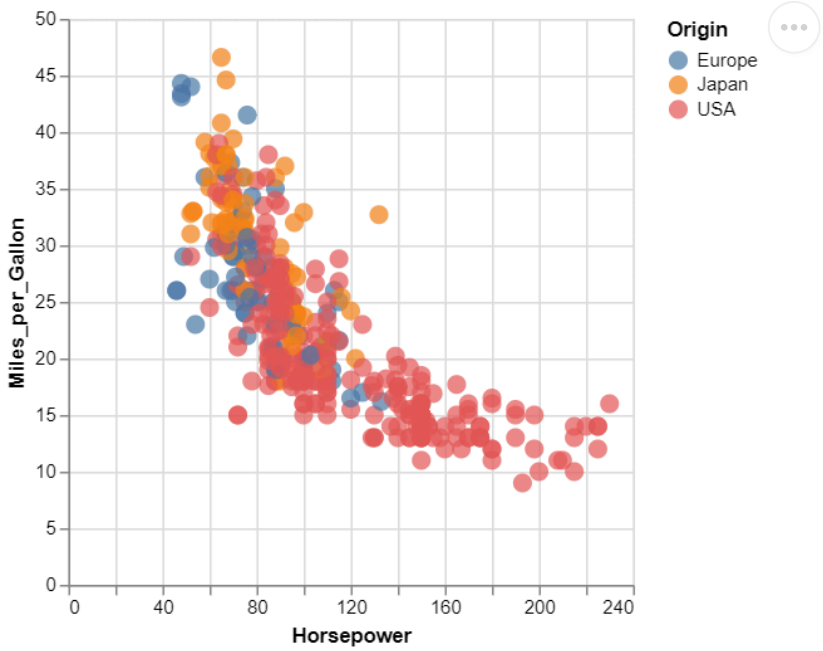
Above we have seen how `alt.condition` can be used to bind the selection to different aspects of the chart. Let's look at a few other ways that a selection can be used:

Binding Scales

For an interval selection, another thing you can do with the selection is bind the selection region to the chart scales:

In [45]:
```
bind = alt.selection_interval(bind='scales')

alt.Chart(cars).mark_circle(size=100).encode(
    x='Horsepower:Q',
    y='Miles_per_Gallon:Q',
    color='Origin:N'
).add_params(
    bind
)
```

Out[45]:



This is essentially what the `chart.interactive()` method does under the hood.

Binding Scales to Other Domains

It is also possible to bind scales to other domains, which can be useful in creating

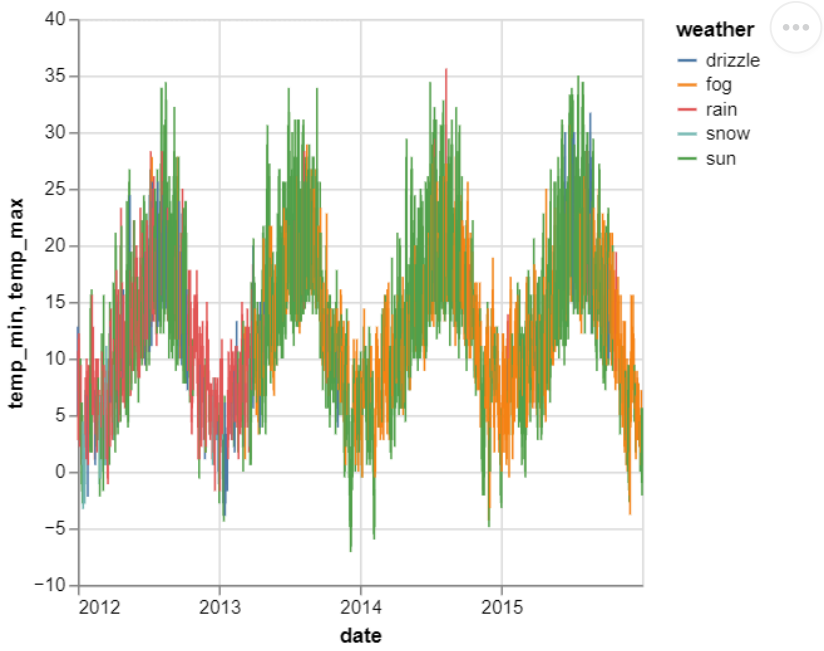
In [46]:
```
weather = data.seattle_weather()
weather.head()
```

Out[46]:

	date	precipitation	temp_max	temp_min	wind	weather
0	2012-01-01	0.0	12.8	5.0	4.7	drizzle
1	2012-01-02	10.9	10.6	2.8	4.5	rain
2	2012-01-03	0.8	11.7	7.2	2.3	rain
3	2012-01-04	20.3	12.2	5.6	4.7	rain
4	2012-01-05	1.3	8.9	2.8	6.1	rain

In [47]:
```
base = alt.Chart(weather).mark_rule().encode(
    x='date:T',
    y='temp_min:Q',
    y2='temp_max:Q',
    color='weather:N'
)

base
```

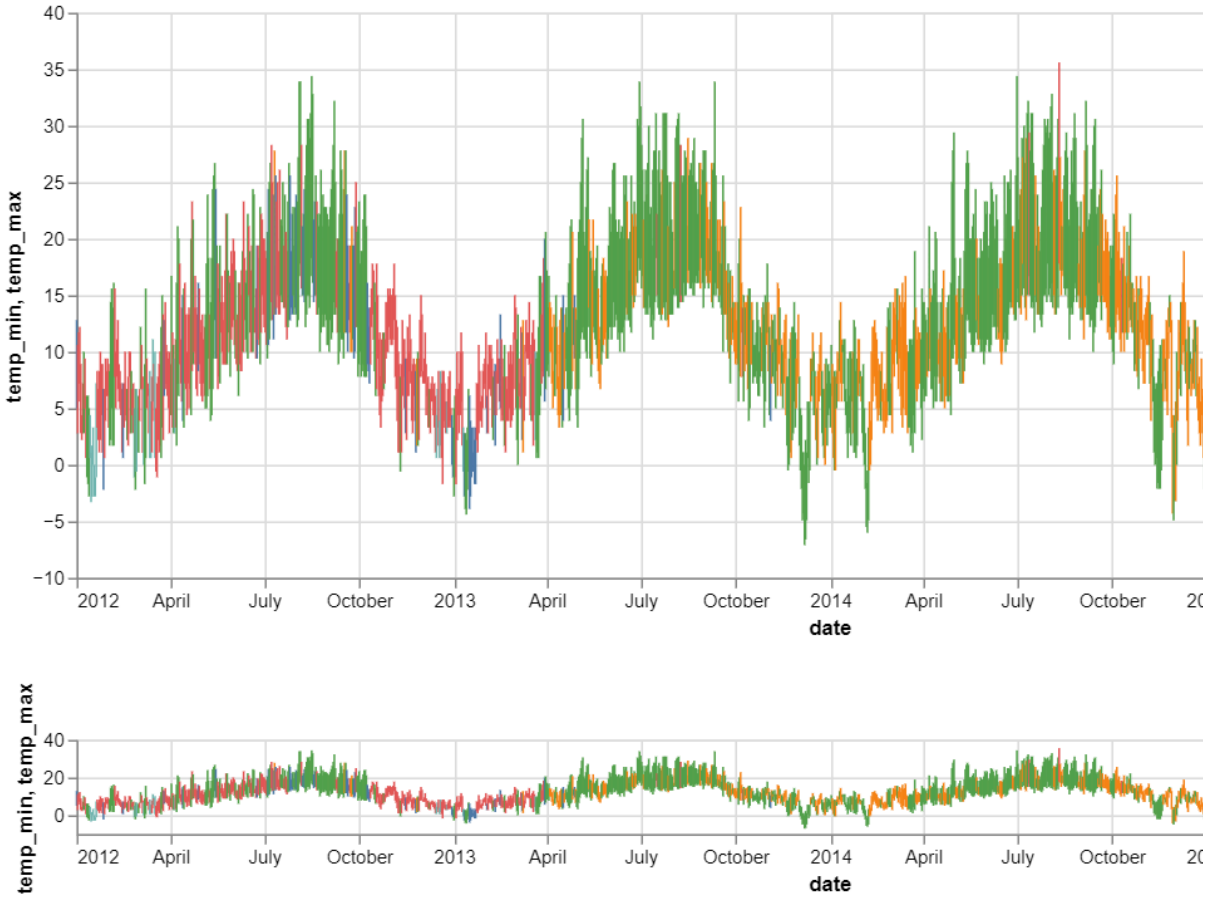
Out[47]:



In [48]:

```
chart = base.properties(
    width=800,
    height=300
)

view = chart.properties(
    width=800,
    height=50
)

chart & view
```

Out[48]:

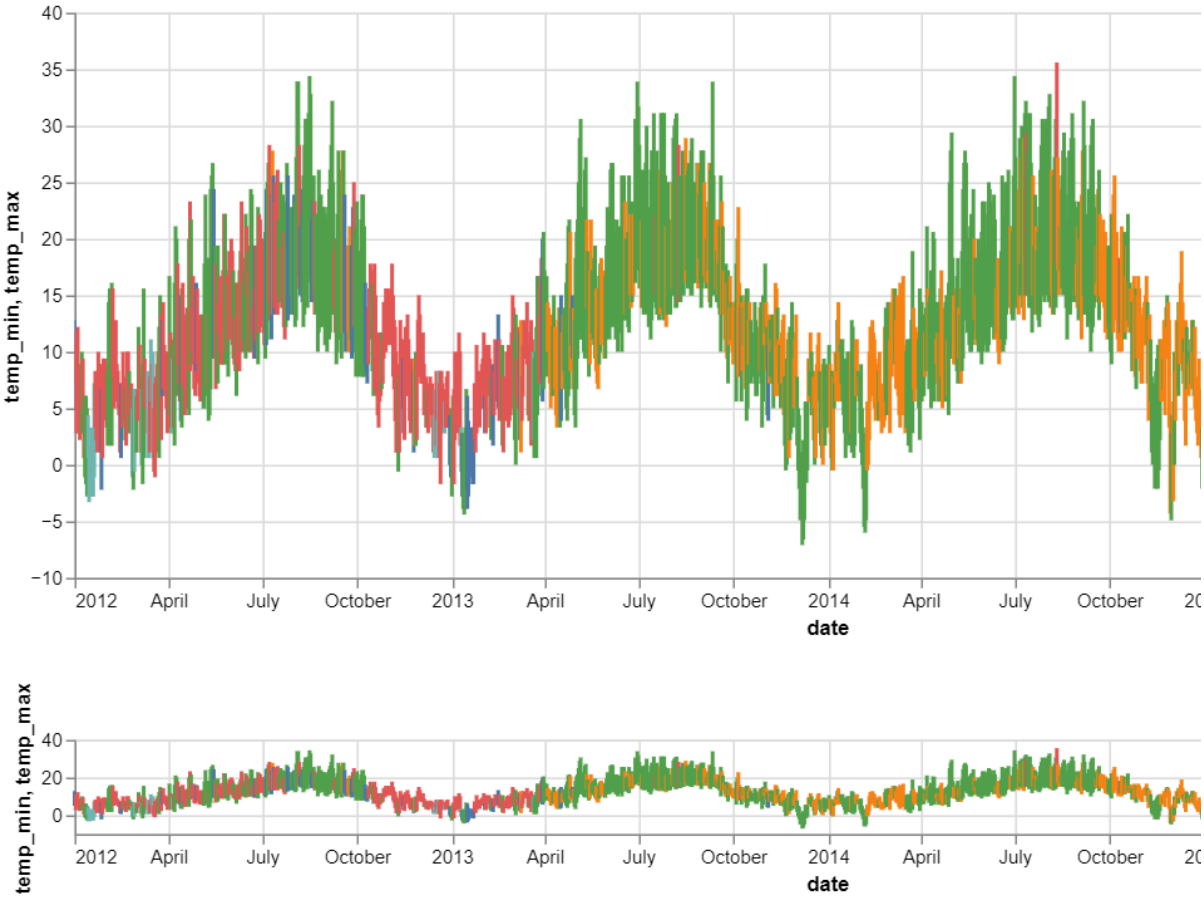


Let us add an interval selection to the bottom chart that will control the domain of the top chart:

In [49]:

```
interval = alt.selection_interval(encodings=['x'])

base = alt.Chart(weather).mark_rule(size=2).encode(
    x='date:T',
    y='temp_min:Q',
    y2='temp_max:Q',
    color='weather:N'
)

chart = base.encode(
    x=alt.X('date:T', scale=alt.Scale(domain=interval))
).properties(
    width=800,
    height=300
)

view = base.add_params(
    interval
).properties(
    width=800,
    height=50,
)

chart & view
```

Out[49]:



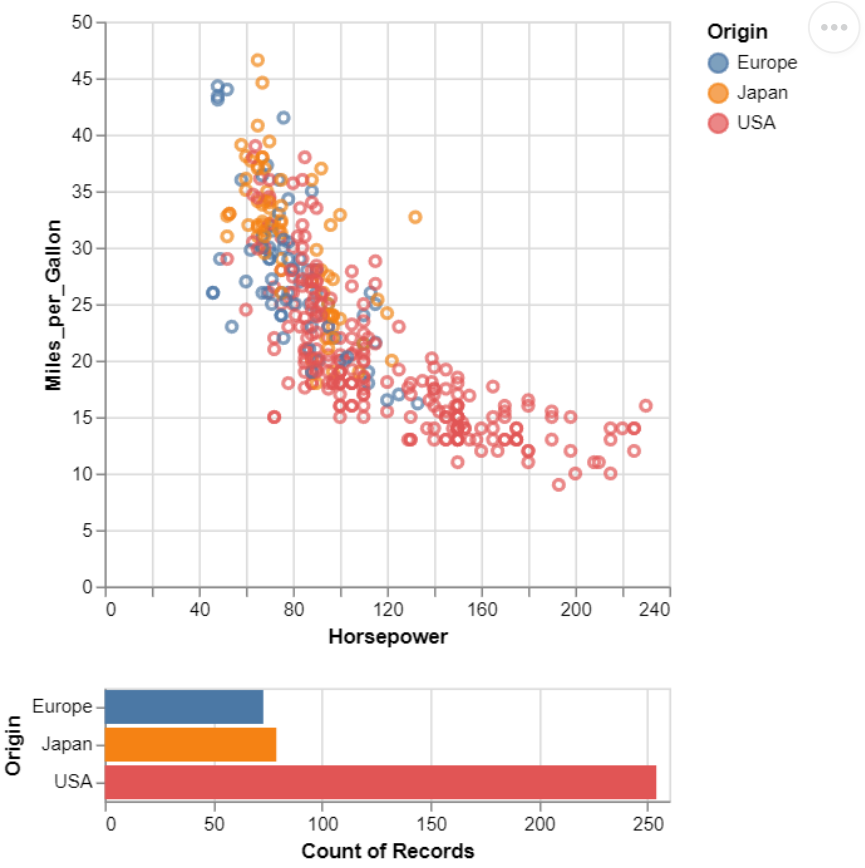
Filtering by Selection / Brushing and Linking

In multi-panel charts, we can use the result of the selection to filter other views of the data. For example, here is a scatter-plot along with a histogram

In [50]:

```
interval = alt.selection_interval()

scatter = alt.Chart(cars).mark_point().encode(
    x='Horsepower:Q',
    y='Miles_per_Gallon:Q',
    color=alt.condition(interval, 'Origin:N', alt.value('lightgray'))
).add_params(
    interval
)

hist = alt.Chart(cars).mark_bar().encode(
    x='count()',
    y='Origin',
    color='Origin'
).transform_filter(
    interval
)

scatter & hist
```

Out[50]:

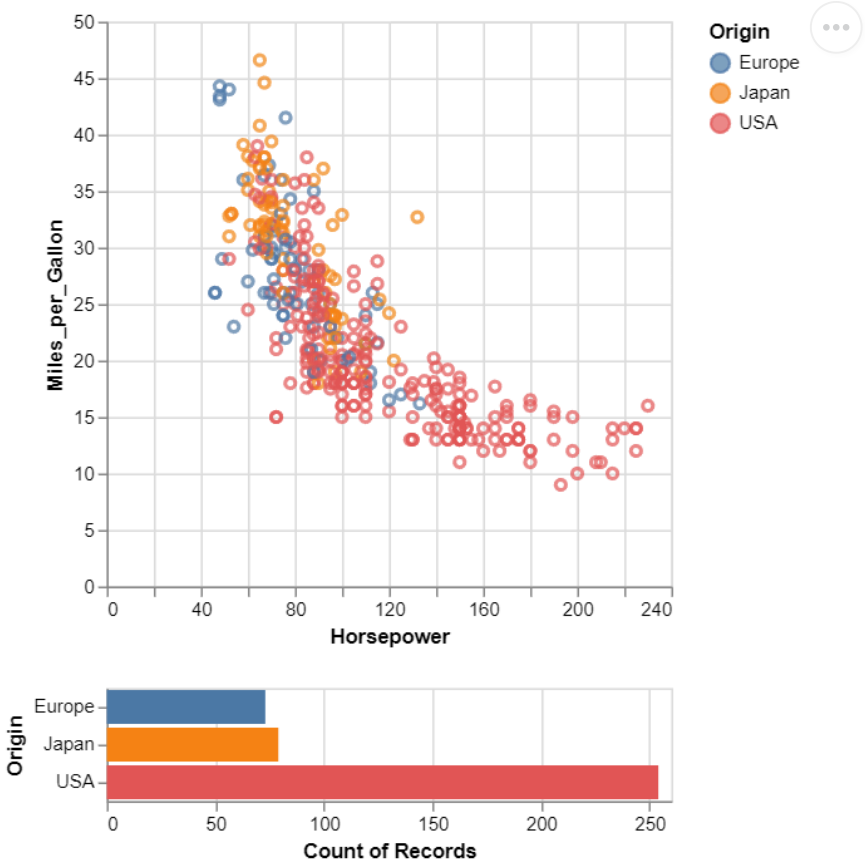


Similarly, you can use a Multi selection to go the other way (allow clicking on the bar chart to filter the contents of the scatter plot. We'll add this to the previous chart:

In [51]:

```
click = alt.selection_point(encodings=['color'])

scatter = alt.Chart(cars).mark_point().encode(
    x='Horsepower:Q',
    y='Miles_per_Gallon:Q',
    color='Origin:N'
).transform_filter(
    click
)

hist = alt.Chart(cars).mark_bar().encode(
    x='count()',
    y='Origin',
    color=alt.condition(click, 'Origin', alt.value('lightgray'))
).add_params(
    click
)

scatter & hist
```

Out[51]:



Selections Summary

- Selection Types:
  - `selection_interval()`
  - `selection_single()`
  - `selection_multi()`
- Bindings
  - bind scales: drag & scroll to interact with plot
  - bind scales on another chart
  - conditional encodings (e.g. color, size)
  - filter data

Exercise

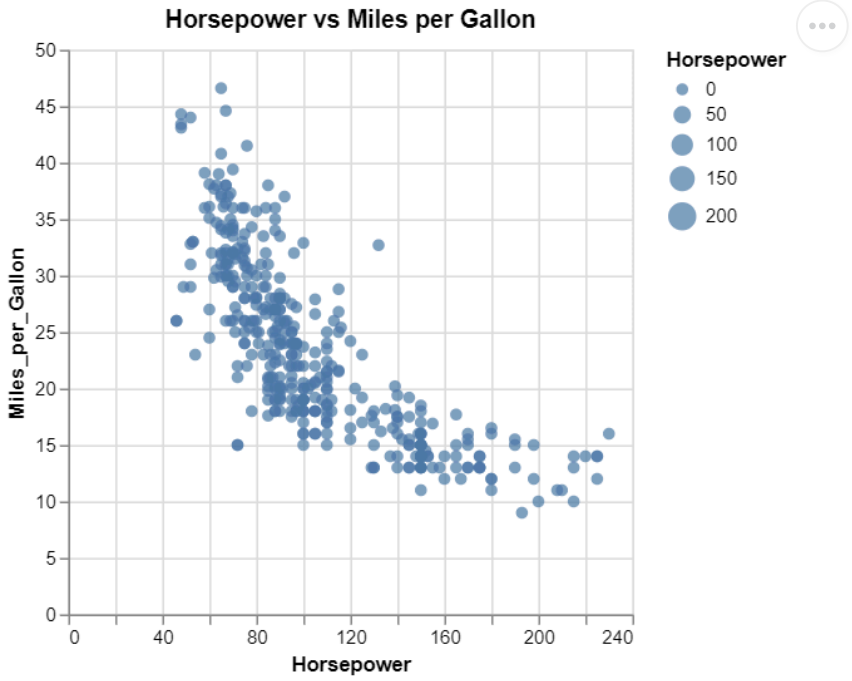
Here you have a chance to try this yourself! Use the interactive examples above to create:

6. Using the cars data, create a scatter-plot of horsepower vs miles per gallon where the *size* of the points becomes larger as you hover over them. **(15 marks)**

7. Using the cars data, create a two-panel histogram (miles per gallon counts in one panel, horsepower counts in the other) where you can drag your mouse to select data in the left panel to filter the data in the second panel. **(15 marks)**

In [ ]:
```
# your code here for scatter-plot

cars.info()
cars.describe()

hover = alt.selection_point(on='mouseover', empty='none')

alt.Chart(cars).mark_circle().encode(
    x='Horsepower:Q',
    y='Miles_per_Gallon:Q',
    size=alt.Size('Horsepower:Q', scale=alt.Scale(range=[40, 250])),
    tooltip=['Name:N', 'Horsepower:Q', 'Miles_per_Gallon:Q']
).add_params(
    hover
).encode(
    size=alt.condition(hover, alt.Size('Horsepower:Q', scale=alt.Scale(range=[40, 250])),
).properties(
    title="Horsepower vs Miles per Gallon"
)
```

Out[ ]:



In [88]:
```
brush = alt.selection_interval(encodings=['x'])

mpg_graph = alt.Chart(cars).mark_bar().encode(
    alt.X('Miles_per_Gallon:Q', bin=True, title='Miles per Gallon'),
    alt.Y('count():Q', title='Count'),
    color=alt.value('#F4D03F')
).add_params(
    brush
).properties(
    title="Histogram of Miles per Gallon"
)

# Right Panel: Histogram for Horsepower with data filtered by the brush on the left panel
hp_graph = alt.Chart(cars).mark_bar().encode(
    alt.X('Horsepower:Q', bin=True, title='Horsepower'),
    alt.Y('count():Q', title='Count'),
    color=alt.value('#7D3C98')
```

```
).transform_filter(
    brush
).properties(
    title="Histogram of Horsepower"
)

# Combine the two histograms in a horizontal layout
final_chart = alt.hconcat(mpg_graph, hp_graph)

# Display the plot
final_chart
```

Out[88]: