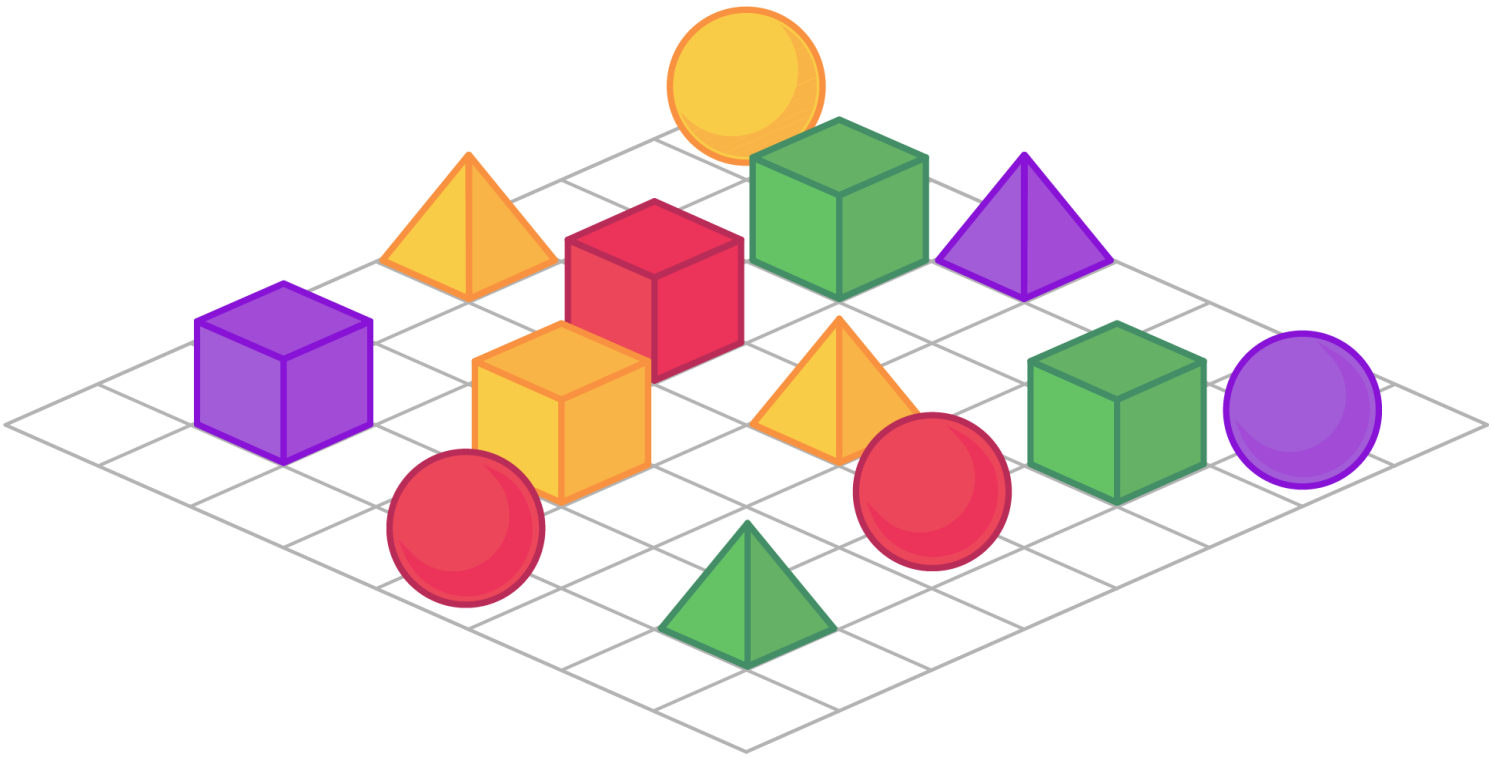
Article

Creating a chart using Swift Charts

Make a chart by combining chart building blocks in SwiftUI.

Overview

Help people understand complex data by focusing on what you want to communicate and who you're communicating to. For example, suppose that you have a collection of colorful toy shapes:



You can formulate questions about this data that you'd like to answer, like which toy shape appears the most, or how many toys are green? One way to represent your data is to collect it into a table. A table organizes the data into columns and rows so it can be easily inspected. For example, you can record how many shapes of each color you have:

	Cube	Sphere	Pyramid	Total
Pink	1	2	0	3
Yellow	1	1	2	4
Purple	1	1	1	3
Green	2	0	1	3
Total	5	4	4	13

However, in many cases, a more effective data representation is a chart. A chart allows you to communicate large amounts of information all at once. The kind of chart that you create and the way you configure the chart depend on what you want to show. To create a chart with Swift Charts, define your data and initialize a `Chart` view with marks and data properties. Then use modifiers to customize different components of the chart, like the legend, axes, and scale.

Define the data source

Think about a chart as an answer to your questions. Suppose you want to know which toy shape appears the most. Start by visualizing how many of each shape you have. To display this information with a chart, create a `ToyShape` structure that represents the information that you want to visualize:

```
struct ToyShape: Identifiable {
    var type: String
    var count: Double
    var id = UUID()
}
```

Then, initialize an array of `ToyShape` structures to hold the data from the table:

```
var data: [ToyShape] = [
    .init(type: "Cube", count: 5),
    .init(type: "Sphere", count: 4),
    .init(type: "Pyramid", count: 4)
```

```
]
```

In a real app, you might download this data from a network connection, or load it from a file.

Initialize a chart view and create marks

Create a `Chart` view that serves as a container for the data series that you want to draw:

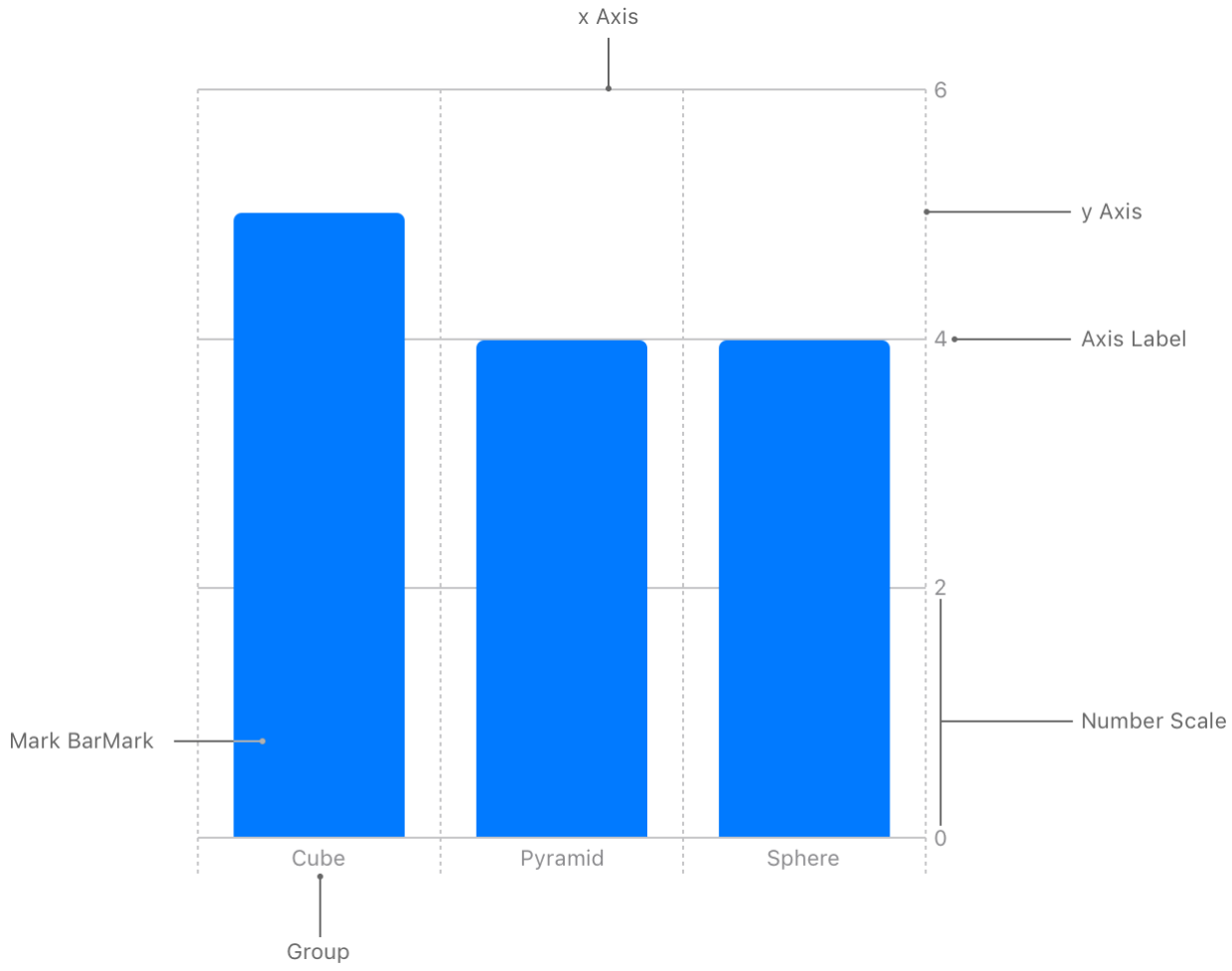
```
import SwiftUI
import Charts

struct BarChart: View {
    var body: some View {
        Chart {
            // Add marks here.
        }
    }
}
```

Inside the chart, specify the graphical marks that represent the data. You can populate it with a variety of kinds of marks, like `BarMark`, `PointMark` or `LineMark`, that plot your data. The kind of mark that you choose depends on how you want to visualize the data. For example, you can use `LineMark` to create a line chart or `PointMark` to produce a scatter plot. In this case, creating a bar chart is a good way to convey the number of each type of toy shape, so you use `BarMark`:

```
Chart {
    BarMark(
        x: .value("Shape Type", data[0].type),
        y: .value("Total Count", data[0].count)
    )
    BarMark(
         x: .value("Shape Type", data[1].type),
         y: .value("Total Count", data[1].count)
    )
    BarMark(
         x: .value("Shape Type", data[2].type),
```

```
            y: .value("Total Count", data[2].count)
        )
    }
```

The resulting chart clearly communicates that the cube toy shape appears the most:



Scale determines the position, height, and color of each `BarMark`. As you plot data on the y-dimension, the framework automatically generates axis labels for the y-axis to map the data values. The scale for the y-dimension is adjusted to match the range of totals for the shape's group.

The above code lists each `BarMark` individually. However, for regular, repetitive data, you can use a `ForEach` structure to represent the same chart more concisely:

```
Chart {
    ForEach(data) { shape in
        BarMark(
            x: .value("Shape Type", shape.type),
```

```
            y: .value("Total Count", shape.count)
        )
    }
}
```

Explore additional data properties

The above bar chart shows how much of each type of toy shape there are, but the earlier table separates each toy shape by color as well. A stacked bar chart can visualize not only the amount of each toy shape type, but also the distribution of colors among the shapes. Before you can plot this new information, you need to represent color in your data structure:

```
struct ToyShape: Identifiable {
    var color: String
    var type: String
    var count: Double
    var id = UUID()
}
```

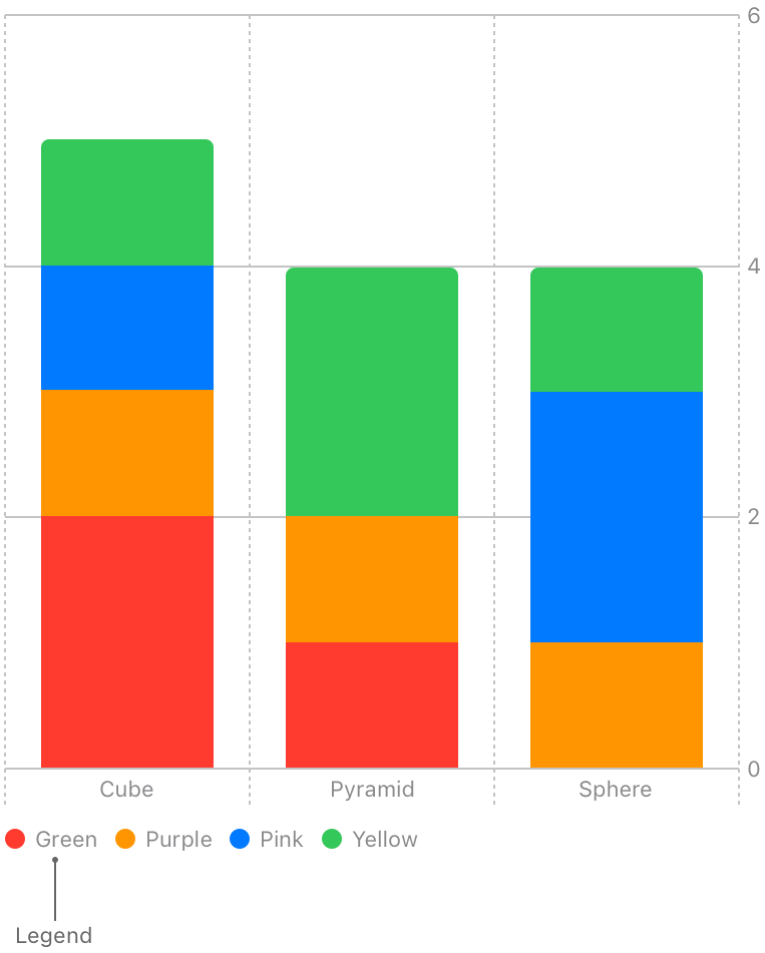
Then update the initialized data to include the color information:

```
var stackedBarData: [ToyShape] = [
    .init(color: "Green", type: "Cube", count: 2),
    .init(color: "Green", type: "Sphere", count: 0),
    .init(color: "Green", type: "Pyramid", count: 1),
    .init(color: "Purple", type: "Cube", count: 1),
    .init(color: "Purple", type: "Sphere", count: 1),
    .init(color: "Purple", type: "Pyramid", count: 1),
    .init(color: "Pink", type: "Cube", count: 1),
    .init(color: "Pink", type: "Sphere", count: 2),
    .init(color: "Pink", type: "Pyramid", count: 0),
    .init(color: "Yellow", type: "Cube", count: 1),
    .init(color: "Yellow", type: "Sphere", count: 1),
    .init(color: "Yellow", type: "Pyramid", count: 2)
]
```

To represent this additional dimension of information, add the `foregroundStyle(by:)` method to the `BarMark`, and indicate the data's color property as the plottable value:

```
Chart {
    ForEach(stackedBarData) { shape in
        BarMark(
            x: .value("Shape Type", shape.type),
            y: .value("Total Count", shape.count)
        )
        .foregroundStyle(by: .value("Shape Color", shape.color))
    }
}
```

The chart now splits the bars into different parts that represent the number of colors for each shape:



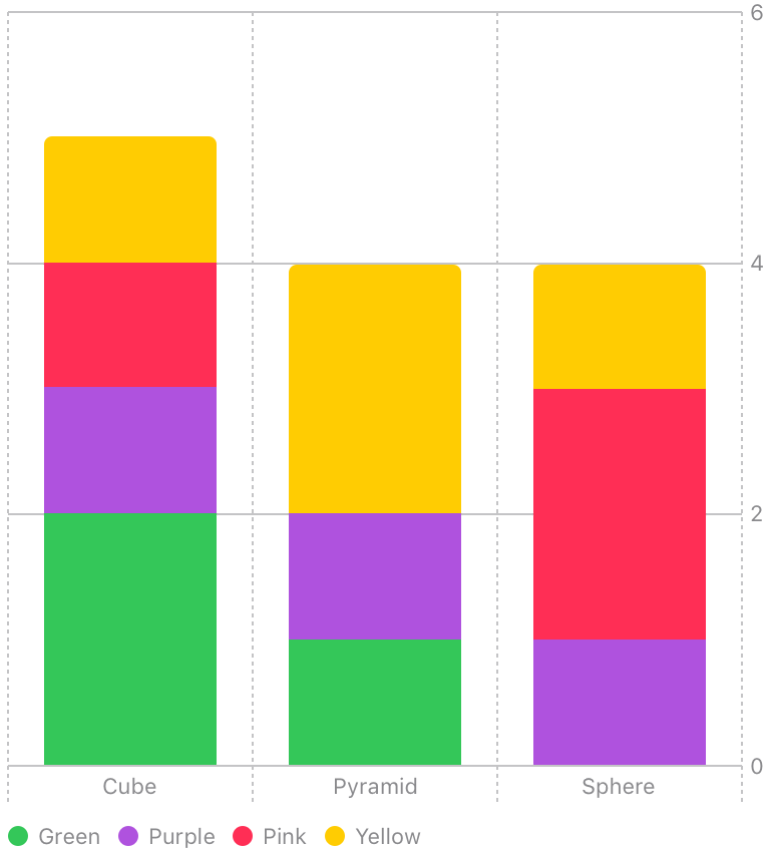
The stacked bar chart chooses colors to represent the new data, and adds a legend to indicate which color represents which kind of data.

Customize your chart

For many charts, the default configuration works well. However, in this case, the colors that the framework assigns to each mark don't match the shape colors that they represent. You can customize the chart to override the default color scale by adding the `Chart/chartForegroundStyleScale(_:)` chart modifier:

```
Chart {
    ForEach(stackedBarData) { shape in
        BarMark(
            x: .value("Shape Type", shape.type),
            y: .value("Total Count", shape.count)
        )
        .foregroundStyle(by: .value("Shape Color", shape.count))
    }
}
.chartForegroundStyleScale([
    "Green": .green, "Purple": .purple, "Pink": .pink, "Yellow": .yellow
])
```

Now the names of the colors match the colors used in the chart, making the chart easier to understand:



This chart makes the relationship between shape counts and colors clear. You can customize charts in many other ways. For example, you can set the bar width, choose different legend symbols, and control the axes.

See Also

Charts

`struct Chart`

A SwiftUI view that displays a chart.

`protocol ChartContent`

A type that represents the content that you draw on a chart.

`struct ChartContentBuilder`

A result builder that you use to compose the contents of a chart.

`struct Plot`

A mechanism for grouping chart contents into a single entity.