

Chart for WPF/Silverlight

2018.02.20 更新

グレースィティ株式会社

目次

| | |
|---|-------|
| 製品の概要 | 7 |
| はじめに | 7 |
| 主な特長 | 7-9 |
| クイックスタート | 9 |
| 手順 1:プロジェクトへの Chart for WPF/Silverlight の追加 | 9-11 |
| 手順 2:グラフへのデータの追加 | 11-16 |
| 手順 3:軸の書式設定 | 16-19 |
| 手順 4:グラフの外観の調整 | 19-20 |
| 重要なヒント | 20-26 |
| XAML クイックリファレンス | 26-27 |
| 例:基本的な折れ線グラフを設定する | 27-28 |
| 例:ガントチャートを設定する | 28-29 |
| 例:積層面グラフを作成する | 29 |
| グラフの要素 | 30 |
| グラフ種別 | 31-32 |
| エリアグラフ | 32-35 |
| 横棒グラフと縦棒グラフ | 35-37 |
| 横棒/縦棒グラフの四角形の角を変更する | 37 |
| 縦棒グラフのマウスクリックイベントの作成 | 37-39 |
| データ系列の各横棒/縦棒の色を指定する | 39 |
| バブルグラフ | 39-40 |
| ローソク足チャート | 40-41 |
| ローソクの幅を変更する | 41-42 |
| HighLowOpenClose チャート | 42-43 |
| コードでの Hi-Low-Open-Close チャートの作成 | 43-44 |
| ガントチャート | 44-47 |
| マークアップでガントチャートの作成 | 47-48 |
| 折れ線グラフ | 48-51 |
| 円グラフ | 51-56 |
| 円グラフに重ならないように接続線を追加する | 56 |
| 円グラフへのラベルの追加 | 56-57 |

| | |
|---|-------|
| すべてのセグメントのオフセットを変更する | 57 |
| 3D 円グラフのデフォルトの表示角度を設定する | 57 |
| ポーラチャート | 57-59 |
| 3D リボングラフ | 59 |
| 多角形グラフ | 59-60 |
| レーダーチャート | 60-61 |
| 階段グラフ | 61-63 |
| 散布グラフ | 63 |
| 散布グラフの作成 | 63-66 |
| 単純なグラフ | 66-71 |
| 特別なチャートタイプと複合チャート | 71 |
| 棒系列と折れ線系列の追加 | 71-72 |
| 縦棒 - 折れ線グラフ | 72 |
| ガウス曲線の作成 | 72-73 |
| パレート図の作成 | 73 |
| グラフ機能 | 74 |
| アニメーション | 74-75 |
| カスタムアニメーションの作成 | 75-76 |
| カスタムアニメーションの作成 | 76 |
| 軸 | 76-78 |
| 軸の注釈 | 78 |
| 軸の注釈の書式 | 78-79 |
| 軸の注釈の回転 | 79-80 |
| 軸のカスタム注釈 | 80-82 |
| カスタム注釈の作成 | 82-83 |
| 軸と目盛りをチャートの反対側に表示する | 83-84 |
| 軸の注釈の書式 | 84-85 |
| 軸の注釈の回転 | 85-86 |
| 軸のカスタム注釈 | 86-88 |
| カスタム注釈の作成 | 88-89 |
| 軸と目盛りをチャートの反対側に表示する | 89-90 |
| 軸の線 | 90 |
| 従属軸 | 90-91 |

| | |
|---|---------|
| 軸の位置 | 91 |
| 軸の範囲 | 91 |
| 軸の原点を設定する | 91-92 |
| 軸のタイトル | 92 |
| 軸の目盛記号 | 92-93 |
| 主目盛のオーバーラップ | 93 |
| 補助目盛のオーバーラップ | 93-94 |
| 主目盛と補助目盛の指定 | 94-95 |
| 主目盛のオーバーラップ | 95-96 |
| 補助目盛のオーバーラップ | 96-97 |
| 主目盛と補助目盛の指定 | 97-98 |
| 軸のグリッド線 | 98 |
| 軸のスクロール | 98-99 |
| グラフの軸の反転と逆転 | 99-100 |
| X 軸と Y 軸を交換する | 100 |
| X 軸と Y 軸を交換する | 100 |
| 複数の軸 | 100-101 |
| 独立した軸を同時に拡大縮小する | 101 |
| 独立した軸を同時に拡大縮小する | 101-102 |
| 軸の対数スケーリング | 102-104 |
| UnitMajor と対数軸 | 104-105 |
| UnitMajor と対数軸 | 105-106 |
| チャート凡例 | 106 |
| チャート凡例を非表示にする | 106 |
| 凡例の方向と位置の変更 | 106 |
| グラフ表示 | 107 |
| プロットの背景の設定 | 107-108 |
| データ連結 | 108 |
| 値のコレクション | 108-109 |
| オブジェクトのコレクション | 109 |
| Observable コレクション | 109-111 |
| データコンテキストのバインディング | 111 |
| double の配列としてのデータコンテキスト | 111 |


| | |
|--|---------|
| Point の配列としてのデータコンテキスト | 111 |
| データ系列のバインディング | 111 |
| 項目名のバインディング | 111-112 |
| X 値のバインディング | 112 |
| DataSet の DataTable へのグラフのバインド | 112-113 |
| データポイントコンバータ | 113-114 |
| データラベル | 114-116 |
| グラフのデータ系列 | 116 |
| さまざまなデータ系列クラス | 116-117 |
| グラフのデータ系列の外観 | 117 |
| DataSeries と XYDataSeries の相違点 | 117 |
| 折れ線グラフまたは円グラフにギャップを表示する | 117-118 |
| グループ化と集計 | 118-119 |
| DataSeries の集計 | 119-120 |
| DateTime のグループ化 | 120-123 |
| カスタムグループ化 | 123-126 |
| エンドユーザー操作 | 126-127 |
| 3D グラフの回転の変更 | 127 |
| 2D でカルトグラフの実行時のインタラクティブ操作の実装 | 127-128 |
| C1Chart でズームする | 128 |
| バブルチャートをズームしながら拡大縮小する | 128-129 |
| マーカーとラベル | 129-130 |
| シンプルな連結マーカー | 130-131 |
| 線とドットのマーカー | 131-132 |
| 十字線マーカー | 132-134 |
| コードでのマーカーの追加 | 134-135 |
| コードでのラベルの更新 | 135 |
| ChartPanel のマウス操作 | 135-136 |
| 複数のプロット領域 | 136-137 |
| プロットエリアのサイズ | 137-138 |
| プロットエリアの外観 | 138 |
| パフォーマンスの最適化 | 138 |

| | |
|--|---------|
| チャートの最適化の有効化 | 138 |
| レンダリングモード | 138-139 |
| バッチ更新の実行 | 139 |
| 設定されているチャートの最適化の無効化 | 139-140 |
| 関数のプロット | 140 |
| コード文字列による関数の定義 | 140 |
| 関数の値の計算 | 140-141 |
| C1Chart の保存とエクスポート | 141 |
| グラフを PDF 形式にエクスポートする | 141 |
| グラフ画像のエクスポート | 141-142 |
| C1Chart を .Png ファイルとして保存する | 142 |
| 系列の生成 | 142-144 |
| MVVM による系列の自動生成 | 144-147 |
| 系列の作成 | 147 |
| レイアウトおよび外観 | 147 |
| グラフのリソースキー | 147-149 |
| グラフスタイル | 149 |
| MouseOver スタイル | 149 |
| MouseOver スタイル | 149-150 |
| テーマ | 150-164 |
| データ系列のカラーパレット | 164-171 |
| プロット要素の色を変更する | 171-172 |
| グラフの書式設定 | 172 |
| XAML の要素 | 172-173 |
| 時系列グラフ | 173-175 |
| 各月の第1日におけるデータラベルの表示 | 175-176 |
| 傾向線 | 176 |
| C1Chart への傾向線の追加 | 176-179 |
| 非回帰傾向線 | 179 |
| チュートリアル | 180 |
| データバインディングのチュートリアル | 180 |
| プログラムによるデータテーブルへのバインド | 180-184 |
| XML へのバインド | 184-188 |

| | |
|---|---------|
| MVVM の使用 | 188 |
| 手順 1: モデルの作成 | 188-190 |
| 手順 2: ビューモデルの作成 | 190 |
| 手順 3: C1Chart を使用したビューの作成 | 190-192 |

製品の概要

Chart for WPF/Silverlight は、さまざまな対話機能を備え、見栄えと訴求力に優れたチャートを WPF/Silverlight アプリケーション上に簡単に表示できるようにします。**Chart for WPF/Silverlight** には **C1Chart** という1つのコントロールが含まれます。このコントロールが、強力なレンダリング、豊富なスタイル設定要素、アニメーション、データ連結などの機能によってチャート表現のレベルを大幅に向上させます。**Chart for WPF/Silverlight** の簡潔さと WPF/Silverlight の機能を組み合わせることで、かつてないほど簡単に、高度にカスタマイズ可能なチャートタイプを作成できます。

 **メモ:** 説明内に含まれるクラスおよびメンバーに対するリファレンスへのリンクは、原則として WPF 版のリファレンスページを参照します。Silverlight 版については、目次から同名のメンバーを参照してください。

はじめに

主な特長

Chart for WPF/Silverlight は、次のユニークな機能を提供します。

- **40 以上のチャートタイプ**

棒グラフ、折れ線グラフ、円グラフ、エリアグラフ、散布図などの標準的な 2D チャートタイプや、ガントチャート、ポーラチャート、HiLo チャート、ローソク足チャートなどの高度なチャートタイプを選択することができます。さらに、ドーナツグラフ、リボングラフ、棒グラフなどの 3D チャートも数多く含まれます。サポートされるチャートタイプの詳細なリストを参照してください。

- **2つのプロパティのみ使用する高度な設計**

Chart では、12 個の組み込みテーマと 22 個の組み込みパレットが提供されます。どちらも、Visual Studio 内で1つのプロパティを使用するだけで設定できます。テーマはチャート領域全体に適用され、パレットはチャート要素(棒、点、円グラフのセグメントなど)にのみ適用されます。テーマをさまざまなパレットと組み合わせることで、多種多様な組み合わせの外観を簡単に作成できます。

- **データラベルとツールチップ**

チャート要素の相対的なデータ値をラベルまたはツールチップとして表示できます。ラベルまたはツールチップとして使用するデータテンプレートを定義して、詳細にカスタマイズできます。

- **対話式操作**

チャートのズーム、スケール、スクロールを可能にすることで、エンドユーザーの使用感が向上しました。マウスドラッグやキー入力など、ユーザーがアクションを実行する方法を指定して、アクションをカスタマイズできます。

- **複数の軸**

Chart では複数の依存軸がサポートされており、Axis オブジェクトを定義してチャートの View.Axes コレクションに追加するだけで、チャートに追加できます。

- **対数軸スケール**

Chart では、任意の底を持つ対数軸スケールがサポートされています。

- **データポイントの選択**

データソースが `ICollectionView` の場合、**C1Chart** コントロールはデータレベルの項目選択をサポートします。ユーザーがプロット要素を選択する方法、選択された要素の外観、添付ラベルの外観を指定できます。

- **アニメーション**

プロット要素のアニメーション API は、プロット要素のさまざまなアニメーション効果を簡単に作成できるようにします。チャートに躍動感を与えるローディングアニメーションを簡単に追加できるようになりました。また、実行時に各プロット

要素を取得しながら、カスタムアニメーションを実行できます。

- **要素レイヤ**

Layers コレクションを使用して、データプロットに重ねて UI 要素を配置することができます。この機能を使用すると、プロットサーフェス上にラベル、吹き出し、マーカーを簡単に表示できます。

- **複合チャート**

複数の異なるチャートタイプを組み合わせて1つのチャートを作成できます。データ系列ごとに異なる視覚化方法を指定して、自在な組み合わせを作成できます。また、軸、凡例、タイトルなど標準的なチャートパーツを共有したまま、複数のプロット領域を水平方向および垂直方向に積み上げることで、データを読み取りやすく、分析しやすく表示することができます。

- **カスタマイズ可能な軸**

C1Chart には、時間表示、対数スケール、軸のスクロール、カスタム軸ラベル、複数軸のサポートなどのさまざまな軸オプションがあります。チャートの Axes コレクションに軸を追加するだけで、チャートの上下左右に複数の依存軸を表示できます。

- **傾向線**

チャート化したデータを傾向線を使用して分析できます。Chart では、多項式、指数、対数、累乗、フーリエ、平均、移動平均、最小、最大などの自動傾向線がサポートされています。

- **照明および影の効果**

照明効果を適用した境界線を作成したり、プロット要素の背後にソフトな影やハードな影を追加できます。

- **積層グラフ**

複雑なデータを簡単な手法で表すことができる積層グラフが提供されています。折れ線グラフ、面グラフ、横棒グラフ、レーダーチャート、プロットチャートを積層型にして、より小さいスペースで複雑なデータを表示できます。

- **動的グラフィック**

Chart では、Silverlight プラットフォームで利用できる投射投影やアニメーションなどの動的なグラフィックを活用できます。

- **集計ビュー**

1つのプロパティを設定するだけで、チャートデータに集計結果を表示できます。集計オプションには、合計、カウント、平均、最小、最大、分散、および標準偏差があります。

- **柔軟なデータ連結**

いくつかのプロパティを設定することで、コントロールをデータテーブル、ビジネスオブジェクトのコレクション、または XML ファイルに連結できます。チャートレベルでデータソースを設定したり、チャート内のデータ系列ごとにデータソースを設定することができます。データ系列、軸、およびプロットエリアを含むチャート構造全体を XAML で宣言的に連結することで、MVVM などの一般的なデザインパターンを利用できます。

- **パフォーマンスの最適化**

C1Chart には、トレンドを示すような大きなデータセットのパフォーマンスを向上させる最適化技術が組み込まれています。C1Chart は、同じデータポイントまたは近接するデータポイント(指定されたピクセル単位の距離に基づく)をレンダリングしないようにして、プロット領域があまり混雑しないように最適化します。折れ線グラフ、面グラフ、および散布図のパフォーマンスを最適化するには、OptimizationRadius を設定します。

- **イメージへのエクスポート**

C1Chart は、Bmp、Png、Gif、Jpeg、Tiff、Wmp などのさまざまな画像形式に直接エクスポートすることができます。それには、SaveImage メソッドを呼び出すだけです。

- **組み込みのテーマとカラーパレット**

Chart for WPF/Silverlight

C1Chart は、2つのプロパティだけで見栄えよく設計することができます。12 種類の組み込みテーマと 22 種類のカラーパレットが用意されています。テーマはチャート領域全体に適用され、パレットはチャート要素(棒、点、円グラフのセグメントなど)にのみ適用されます。テーマをさまざまなパレットと組み合わせることで、多種多様な組み合わせの外観を簡単に作成できます。コードやマークアップで細かくカスタマイズすることもできます。

- **照明および影の効果**

照明効果を適用した境界線を作成したり、プロット要素の背後にソフトな影やハードな影を追加できます。アンビエント照明、指向性照明、点照明、およびスポット照明を適用して光源の効果を変えることで、3D チャート要素の外観を引き立てることができます。

- **チャート凡例**

1つのプロパティを介してチャートに接続する**C1ChartLegend** コントロールを使用して、独立したチャート凡例を作成できます。このような設計により、凡例の配置やスタイル設定の際に最大限の柔軟性を得ることができます。

- **Silverlight Toolkit テーマのサポート (Silverlight のみ)**

12 個の組み込みテーマに加えて、Chart には、Microsoft Silverlight Toolkit の中で最もよく使用されるテーマである ExpressionDark、ExpressionLight、WhistlerBlue、RainierOrange、ShinyBlue、BureauBlack などとも付属しています。

- **XBAP サポート (WPF のみ)**

C1Chart は、Windows Presentation Foundation の XBAP 配布機能と完全な互換性があります。XBAP 配布を使用すると、Windows をインストールすることなく、完全な機能を備えたアプリケーションを(サポートされている)クライアントブラウザに公開できます。

クイックスタート

以下のクイックスタートは、**Chart for WPF/Silverlight** の基本的な使用法の習得を目的としています。このクイックスタートでは、横棒グラフなどの標準的なグラフを作成するための4つの基本的な手順(グラフの選択、グラフへの1つ以上のデータ系列の追加、軸のセットアップと書式設定、**Theme** プロパティを使用したグラフの外観の調整)について、順を追って説明します。

手順 1: プロジェクトへの Chart for WPF/Silverlight の追加

この手順では、Visual Studio または Blend のいずれかにより、**Chart for WPF/Silverlight** を使用してグラフアプリケーションを作成します。**C1Chart** コントロールを Visual Studio または Blend のプロジェクトに追加すると、実用的な縦棒グラフがダミーデータで作成されます。XAML コードのみを使用して Chart コントロールを初期化した場合は、ダミーデータなしで空のグラフが作成されます。

XAML を使用して Chart for WPF/Silverlight を Visual Studio プロジェクトに追加するには、次の作業を実行します。

1. Visual Studio で新しい WPF/Silver プロジェクトを作成します。
2. **C1.WPF.4.dll** または **C1.Silverlight.5.dll** および **C1.WPF.C1Chart.4.dll** または **C1.Silverlight.C1Chart.5.dll** アセンブリへの参照を追加します。ソリューションエクスプローラで、**[参照]**を右クリックして**[参照の追加]**を選択します。**[参照の追加]**ダイアログボックスで、「**参照**」タブを選択します。C1.WPF.4.dll と C1.WPF.C1Chart を参照して、**<OK>**を選択します。
3. **System** と **C1.WPF.C1Chart** のプレフィックスを定義します。

XAML

```
xmlns:System="clr-namespace:System;assembly=microsoft.windows.common-usercore.dll"
Title="Window1" Height="332" Width="536" xmlns:c1chart="clr-namespace:C1.WPF.C1Chart;assembly=C1.WPF.C1Chart">
```

4. グリッド内で、**C1Chart** コントロールを初期化します。

XAML

```
<clchart:C1Chart Margin="0,0,8,8" MinHeight="160" MinWidth="240"
Content="C1Chart">
</clchart:C1Chart>
```

まだデータを追加していないため、グラフの表示は空です。次の「[手順 2: グラフへのデータの追加](#)」では、**C1Chart** のデータを追加します。

Chart for WPF/Silverlight を Blend プロジェクトに追加するには、以下の作業を実行します。

1. Blend で新しい WPF プロジェクトを作成します。
2. **C1Chart** コントロールをウィンドウに追加します。デフォルトのダミーデータが Chart コントロールに追加されます。

XAML

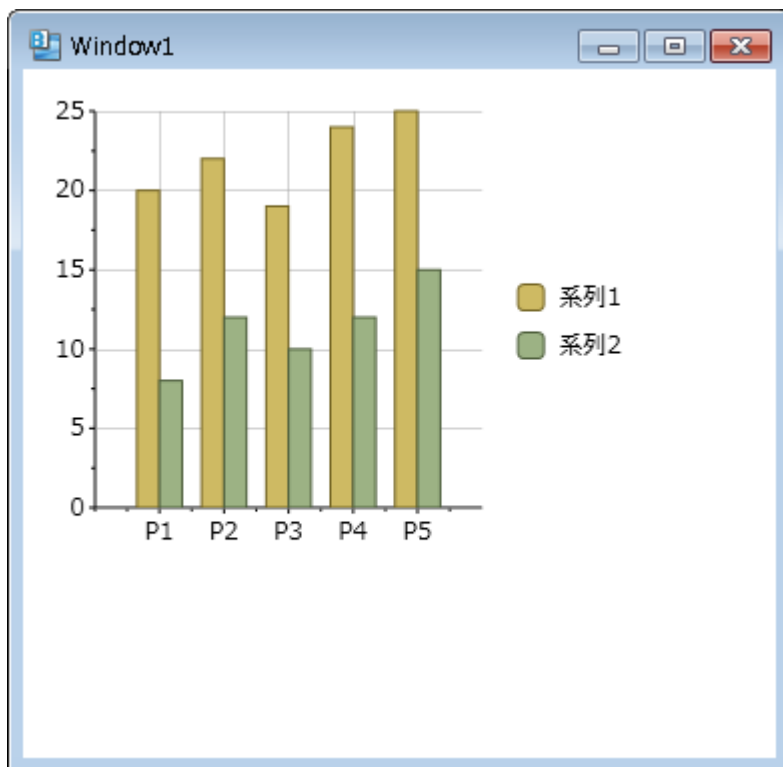
```
<clchart:C1Chart Margin="81,108,67,44">
  <clchart:C1Chart.Data>
    <clchart:ChartData ItemNames="P1 P2 P3 P4 P5">
      <clchart:DataSeries Label="Series 1" RenderMode="Default"
Values="20 22 19 24 25"/>
      <clchart:DataSeries Label="Series 2" RenderMode="Default"
Values="8 12 10 12 15"/>
    </clchart:ChartData>
  </clchart:C1Chart.Data>
  <clchart:C1ChartLegend DockPanel.Dock="Right"/>
</clchart:C1Chart>
```

3. **C1Chart** コントロールをサイズ変更して、ウィンドウ全体に広げます。次の「[手順 2: グラフへのデータの追加](#)」では、**C1Chart** のデータを追加します。

プログラムを実行して、以下を確認します。

Chart コントロールを Blend または Visual Studio のプロジェクトに追加した場合、**C1Chart** コントロールは下の図のように表示されます。XAML コードのみを使用して Chart コントロールを追加した場合は、デフォルトのデータが追加されていないため、グラフの表示は空になります。

Chart for WPF/Silverlight



これで、グラフアプリケーションの作成は完了しました。次の手順では、C1Chart コントロールのデータ系列をカスタマイズします。

手順 2: グラフへのデータの追加

前の手順では、C1Chart コントロールをウィンドウに追加しました。この手順では、DataSeries オブジェクトとそのデータを追加します。

Visual Studio で XAML を使用してデータをグラフに追加するには、以下の作業を実行します。

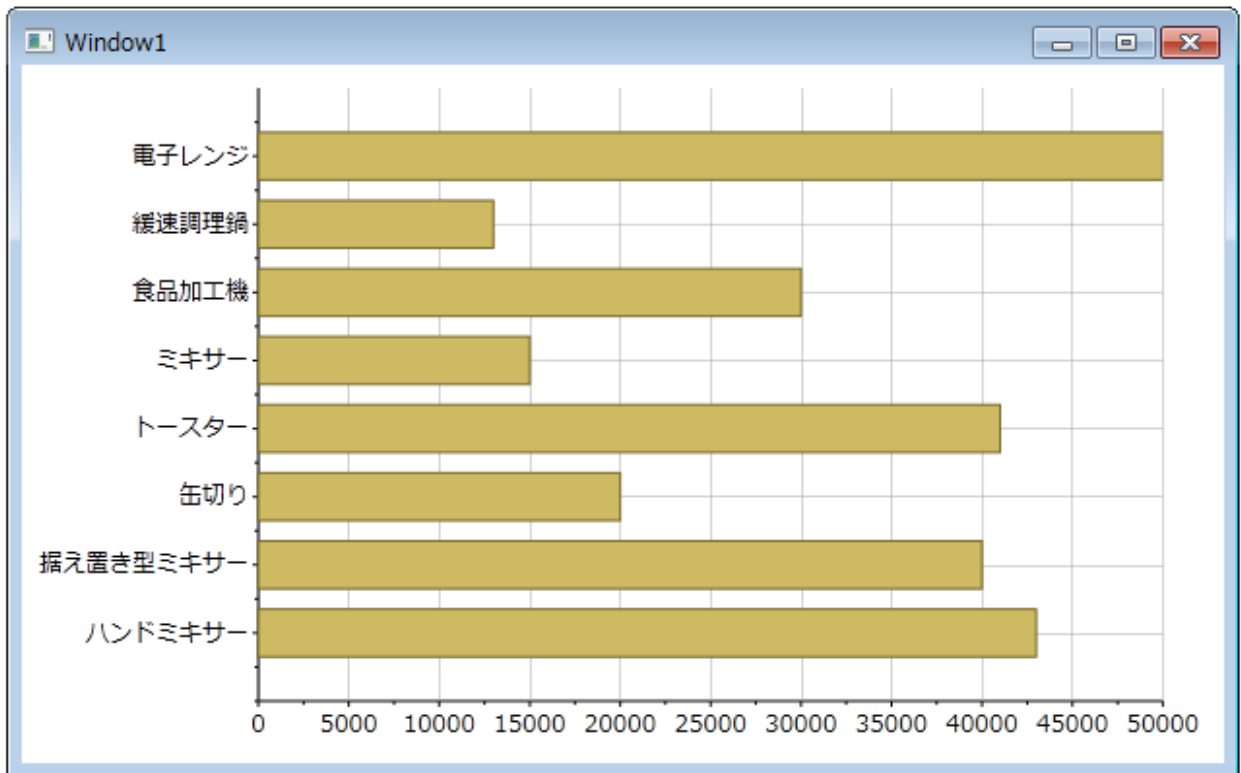
1. 次の XAML コードを使用して、タグ内で **ChartType** プロパティを **Bar** に設定します。
2. ChartType="Bar"
3. 次の XAML コードを使用して、Chart コントロールのデータを追加します。

XAML

```
<c1chart:C1Chart.Data>
  <c1chart:ChartData>
    <c1chart:ChartData.ItemNames>
      <x:Array Type="{x:Type System:String}">
        <System:String>ハンドミキサー</System:String>
        <System:String>据え置き型ミキサー</System:String>
        <System:String>缶切り</System:String>
        <System:String>トースター</System:String>
        <System:String>ミキサー</System:String>
        <System:String>食品加工機</System:String>
        <System:String>緩速調理鍋</System:String>
        <System:String>電子レンジ</System:String>
      </x:Array>
    </c1chart:ChartData.ItemNames>
    <c1chart:DataSeries Values="43000 40000 20000 41000 15000 30000 13000">
```

```
50000" AxisX="価格" AxisY="キッチン家電" />
    </clchart:ChartData>
</clchart:C1Chart.Data>
```

この手順では、8つの X 値を含む **DataSeries** を1つ使用しています。各データ値の文字列名を表すために **String** 型の **ItemNames** を **ChartData** に追加しました。いくつかの項目名にスペースが含まれるため、**ItemNames** に文字列名の配列を使用しました。「[手順 1: プロジェクトへの Chart for WPF の追加](#)」の手順4で名前空間を宣言したため、**System:String** 名前空間を使用できました。新しいデータは、グラフで次のように表示されます。



次の「[手順 3: 軸の書式設定](#)」では、XAML コードを使用して軸をカスタマイズする方法を学びます。

Blend で[プロパティ]ウィンドウを使用してデータをグラフに追加するには、次の作業を実行します。

1. ウィンドウで **C1Chart** コントロールを選択してアクティブにし、[プロパティ]ウィンドウの「**外観**」タブに移動して、**ChartType** プロパティを「**Bar**」に設定します。
2. [プロパティ]ウィンドウで「**その他**」タブに移動します。
3. 「**その他**」タブで、**Data (ChartData)** を見つけて「**新規作成**」ボタンをクリックします。次のように、**ChartData** オブジェクトが XAML コードに追加されます。

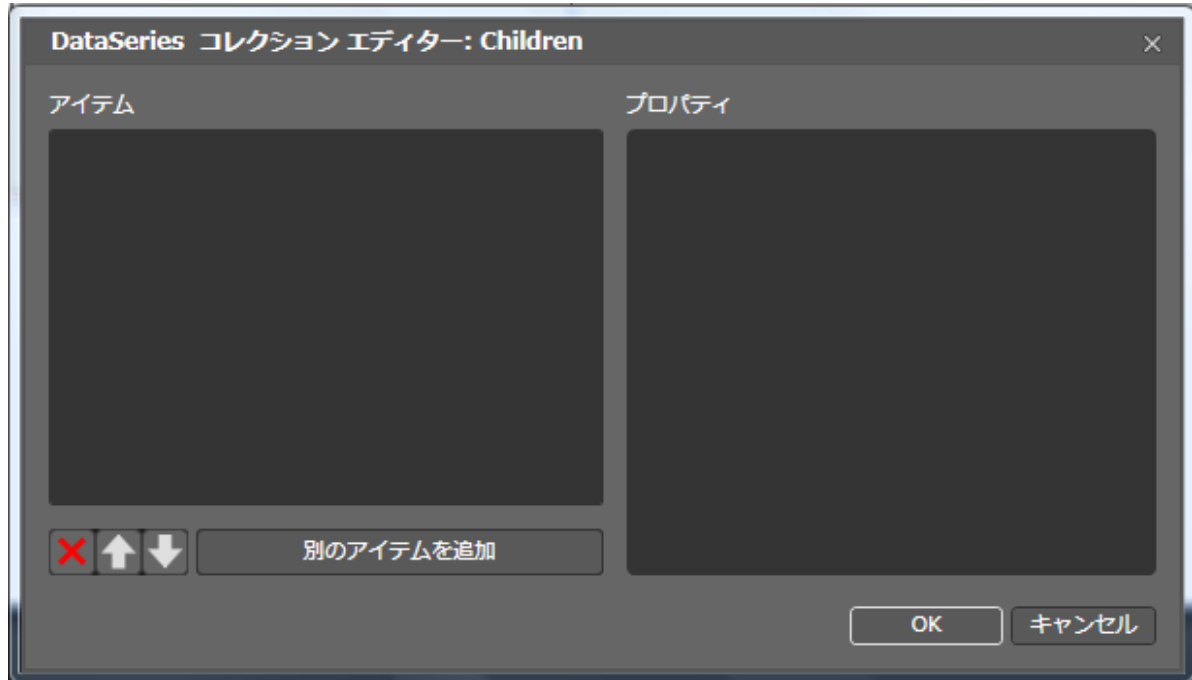
XAML

```
<clchart:C1Chart ChartType="Bar" Margin="108,74,55,66">
    <clchart:C1Chart.Data>
        <clchart:ChartData/>
    </clchart:C1Chart.Data>
    <clchart:C1ChartLegend DockPanel.Dock="Right"/>
</clchart:C1Chart>
```

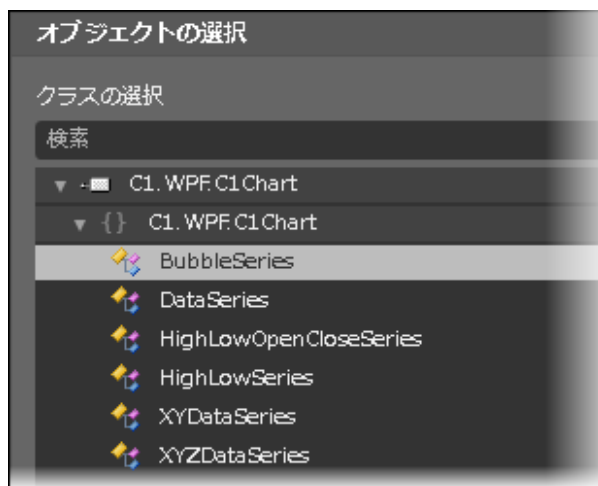
ダミーデータが削除され、まだデータが追加されていないため、**C1Chart** コントロールの表示は空になります。

4. **C1Chart** の[プロパティ]ウィンドウで **Data (ChartData)** の横にある矢印をクリックして、**ChartData** プロパティを拡張します。**Children (コレクション)** プロパティの横にある「**...**」ボタンをクリックします。[**DataSeriesコレクションエディター**：

[Children]ダイアログボックスが表示されます。



5. <別のアイテムを追加>ボタンをクリックして、いずれかの型のデータ系列を**DataSeriesCollection**に追加します。[オブジェクトの選択]ダイアログボックスが表示されます。



[オブジェクトの選択]ダイアログボックスでは、作成するグラフのタイプに応じて、グラフオブジェクトの **BubbleSeries**、**DataSeries**、**HighLowOpenCloseSeries**、**HighLowSeries**、**XYDataSeries**、**XYZDataSeries** から1つを選択できます。横棒グラフを作成するには、**DataSeries** を使用します。**DataSeries** オブジェクトを選択したら、それは **DataSeriesCollection** に追加されます。複数の系列を追加するには、<別のアイテムを追加>ボタンをクリックできます。

6. [オブジェクトの選択]ダイアログボックスから**DataSeries**を選択して、<OK>をクリックします。
[0] **DataSeries** オブジェクトが[アイテム]パネルに追加されます。
7. 次に各製品の価格を表す値を追加します。プルパティ ウィンドウで**Value** を見つけて、43000 40000 20000 41000 15000 30000 13000 50000 を入力します。終了したら<OK>をクリックして、[**DataSeries コレクションエディター: Children**]ダイアログボックスを閉じます。

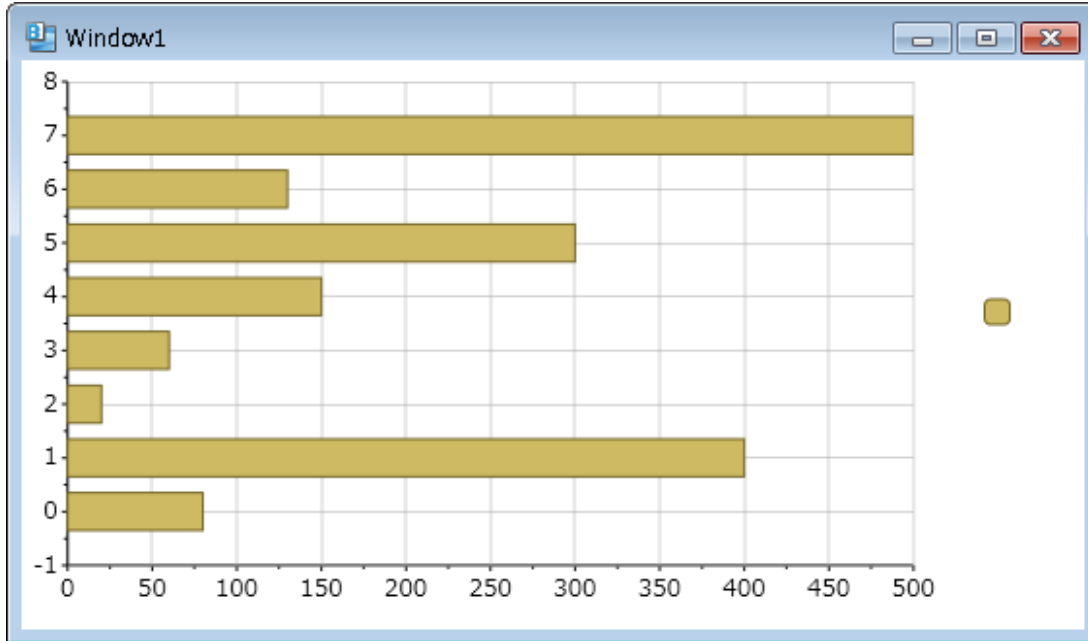
XAML

```
<clchart:C1Chart.Data>
  <clchart:ChartData>
    <clchart:DataSeries Values="43000 40000 20000 41000 15000 30000
```

```
13000 50000"/>
    </clchart:ChartData>
</clchart:C1Chart.Data>
```

XAML コードは次のようになります。

8. プロジェクトを実行します。次のように、1つの系列を含む単純な横棒グラフが表示されます。



9. 「Data(ChartData)」ノードを拡張して、ItemNamesプロパティを見つけます。名前として、ハンドミキサー、据え置き型ミキサー、缶切り、トースター、ミキサー、食品加工機、緩速調理鍋、および電子レンジを、この順番で入力します。[Enter]キーを押して戻り、必要に応じてXAMLコードに変更を加えます。XAMLコードは、次のようになるはずです。

次の「[手順 3: 軸の書式設定](#)」では、XAMLコードを使用して軸をカスタマイズする方法を学びます。

XAML

```
<clchart:ChartData.ItemNames>
    <x:Array Type="{x:Type System:String}">
        <System:String>ハンドミキサー</System:String>
        <System:String>据え置き型ミキサー</System:String>
        <System:String>缶切り</System:String>
        <System:String>トースター</System:String>
        <System:String>ミキサー</System:String>
        <System:String>食品加工機</System:String>
        <System:String>緩速調理鍋</System:String>
        <System:String>電子レンジ</System:String>
    </x:Array>
</clchart:ChartData.ItemNames>
```

コードビハインドファイルでプログラムによりデータをグラフに追加するには、次の作業を実行します。

1. Visual Studio または Blend で新しい WPF プロジェクトを作成します。
2. C1Chart コントロールを Window1 に追加します。
3. Window1 を右クリックして[コードの表示]を選択し、エディタを開きます。
4. C1.WPF.C1Chart 名前空間の指示文を追加します。

WPF

Chart for WPF/Silverlight

Visual Basic

```
Imports Cl.WPF.ClChart
```

C#

```
using Cl.WPF.ClChart;
```

Silverlight

Visual Basic

```
Imports Cl.Silverlight.Chart
```

C#

```
using Cl.Silverlight.Chart;
```

5. コンストラクタの Window1 クラスで次のコードを追加して、横棒グラフを作成します。

Visual Basic

```
' 既存のデータをクリア
c1Chart1.Data.Children.Clear()
' データを追加
Dim ProductNames As String() = {"ハンドミキサー", "据え置き型ミキサー", "缶切り", "トースター",
"ミキサー", "食品加工機", _
"緩速調理鍋", "電子レンジ"}
Dim PriceX As Integer() = {43000, 40000, 20000, 41000, 15000, 30000, _
13000, 50000}
' 製品価格の系列を1つ作成
Dim ds1 As New DataSeries()
ds1.Label = "価格"
' 価格データを設定
ds1.ValuesSource = PriceX
' 系列をグラフに追加
c1Chart1.Data.Children.Add(ds1)
' 項目名を追加
c1Chart1.Data.ItemNames = ProductNames
' グラフタイプを設定
c1Chart1.ChartType = ChartType.Bar
```

C#

```
// 既存のデータをクリア
c1Chart1.Data.Children.Clear();
// データを追加
string[] ProductNames = {"ハンドミキサー", "据え置き型ミキサー", "缶切り", "トースター", "ミキ
サー", "食品加工機", "緩速調理鍋", "電子レンジ"};
int[] PriceX = { 43000, 40000, 20000, 41000, 15000, 30000, 13000, 50000 };
// 製品価格の系列を1つ作成
```



```

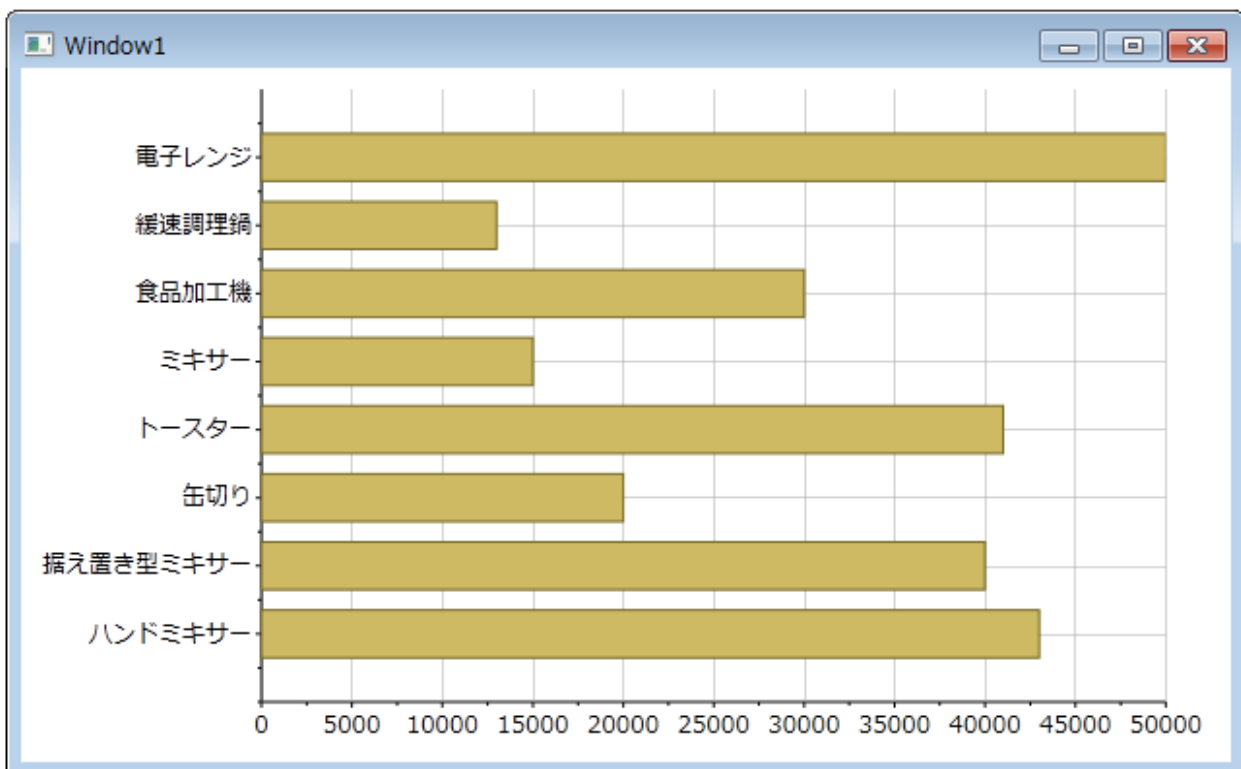
    DataService ds1 = new DataService();
    ds1.Label = "価格";
    // 価格データを設定
    ds1.ValuesSource = PriceX;
    // 系列をグラフに追加
    c1Chart1.Data.Children.Add(ds1);
    // 項目名を追加
    c1Chart1.Data.ItemNames = ProductNames;
    // グラフタイプを設定
    c1Chart1.ChartType = ChartType.Bar;

```

次の「手順 3:軸の書式設定」では、プログラムで軸をカスタマイズする方法を学びます。

プログラムを実行して、以下を確認します。

次のように、文字列値が Y 軸に表示されます。



次の一連の手順では、軸をカスタマイズします。

これで、Chart コントロールへのデータの追加は完了しました。次の手順では、軸を書式設定します。

手順 3:軸の書式設定

この手順では、**ChartView** オブジェクトを追加して、X 軸をカスタマイズできるようにします。

Visual Studio または **Blend** で **XAML** を使用して **Chart for WPF/Silverlight** の軸を書式設定するには、以下の作業を行います。

1. **ChartView** オブジェクトを追加して、X 軸と Y 軸のタイトルを設定できるようにします。**ChartView** オブジェクトは、データが含まれるグラフの領域(軸を含む)を表します。グラフの軸の詳細については、「[軸](#)」を参照してください。軸のタイトルは単なるテキストではなく、UIElement オブジェクトです。この例では、**TextBlock** 要素を使用して、X 軸と Y 軸のタイトルにテキストを割り当てます。**TextBlock** 要素を追加したら、その前景色を変更して中央揃えにすることによって、テキストを書式設定できます。

XAML

```
<clchart:C1Chart >
  <clchart:C1Chart.View>
    <clchart:ChartView>
      <clchart:ChartView.AxisX>
        <clchart:Axis>
          <clchart:Axis.Title>
            <TextBlock Text="価格" TextAlignment="Center"
Foreground="Crimson"/>
          </clchart:Axis.Title>
        </clchart:Axis>
      </clchart:ChartView.AxisX>
      <clchart:ChartView.AxisY>
        <clchart:Axis>
          <clchart:Axis.Title>
            <TextBlock Text="キッチン家電" TextAlignment="Center"
Foreground="Crimson"/>
          </clchart:Axis.Title>
        </clchart:Axis>
      </clchart:ChartView.AxisY>
    </clchart:ChartView>
  </clchart:C1Chart.View>
</clchart:C1Chart>
```

2. デフォルトの **AxisX.MajorUnit** の単位値を 5000 から 2000 に変更します。これで、**View** オブジェクトの XAML コードは、次のようになるはずですが。

XAML

```
<clchart:C1Chart.View>
  <clchart:ChartView>
    <clchart:ChartView.AxisX>
      <clchart:Axis Min="0" Max="50000" MajorUnit="2000" >
        <clchart:Axis.Title>
          <TextBlock Text="価格" TextAlignment="Center"
Foreground="Crimson" />
        </clchart:Axis.Title>
      </clchart:Axis>
    </clchart:ChartView.AxisX>
    <clchart:ChartView.AxisY>
      <clchart:Axis>
        <clchart:Axis.Title>
          <TextBlock Text="キッチン家電"
TextAlignment="Center" Foreground="Crimson" />
        </clchart:Axis.Title>
      </clchart:Axis>
    </clchart:ChartView.AxisY>
  </clchart:ChartView>
</clchart:C1Chart.View>
```

3. **ChartView.AxisX** オブジェクトの内 `<clchart:Axis>` `</clchart:Axis>` で、**AnnoFormat** を設定して x 軸の数値の x 値を通貨に変更し、**AnnoAngle** プロパティを設定して X 軸の注釈を反時計回りに 60° 回転させます。

```
<c1chart:Axis AnnoFormat="c" AnnoAngle="60" />
```

4. ChartView.AxisYオブジェクトの内<c1chart:Axis></c1chart:Axis>で、**Reversed**プロパティをTrueに設定して、Y軸の方向を逆転させます。次の「[手順 4: グラフの外観の調整](#)」では、XAMLを使用してグラフの外観をカスタマイズする方法を学びます。

コードビハインドファイルでプログラムにより Chart for WPF/Silverlight の軸を書式設定するには、次の作業を実行します。

コンストラクタの Window1 クラスで次のコードを追加して、グラフの軸を書式設定します。

Visual Basic

' 軸のタイトルを設定

```
C1Chart1.View.AxisY.Title = New TextBlock() With {.Text = "キッチン家電",
.TextAlignment = TextAlignment.Center, .Foreground =
System.Windows.Media.Brushes.Crimson}
```

```
C1Chart1.View.AxisX.Title = New TextBlock() With {.Text = "価格", .TextAlignment =
TextAlignment.Center, .Foreground = System.Windows.Media.Brushes.Crimson}
```

' 軸ラベルを反転順番に設定

```
c1Chart1.View.AxisY.Reversed = True
```

' 軸の範囲を設定

```
C1Chart1.View.AxisX.Min = 0
```

```
C1Chart1.View.AxisX.Max = 50000
```

' ラベル単位を設定

```
c1Chart1.View.AxisX.MajorUnit = 2000
```

' 金融向けの書式設定

```
C1Chart1.View.AxisX.AnnoFormat = "c"
```

' 軸の注釈の回転

```
C1Chart1.View.AxisX.AnnoAngle = "60"
```

C#

// 軸のタイトルを設定

```
c1Chart1.View.AxisY.Title= new TextBlock() { Text = "キッチン家
電",TextAlignment=TextAlignment.Center, Foreground =
System.Windows.Media.Brushes.Crimson };
```

```
c1Chart1.View.AxisX.Title = new TextBlock() { Text = "価格",
TextAlignment=TextAlignment.Center, Foreground = System.Windows.Media.Brushes.Crimson
};
```

// 軸ラベルを反転順番に設定

```
c1Chart1.View.AxisY.Reversed = true;
```

// 軸の範囲を設定

```
c1Chart1.View.AxisX.Min = 0;
```

```
c1Chart1.View.AxisX.Max = 50000;
```

// ラベル単位を設定

```
c1Chart1.View.AxisX.MajorUnit = 2000;
```

// 金融向けの書式設定

```
c1Chart1.View.AxisX.AnnoFormat = "c";
```

// 軸の注釈の回転

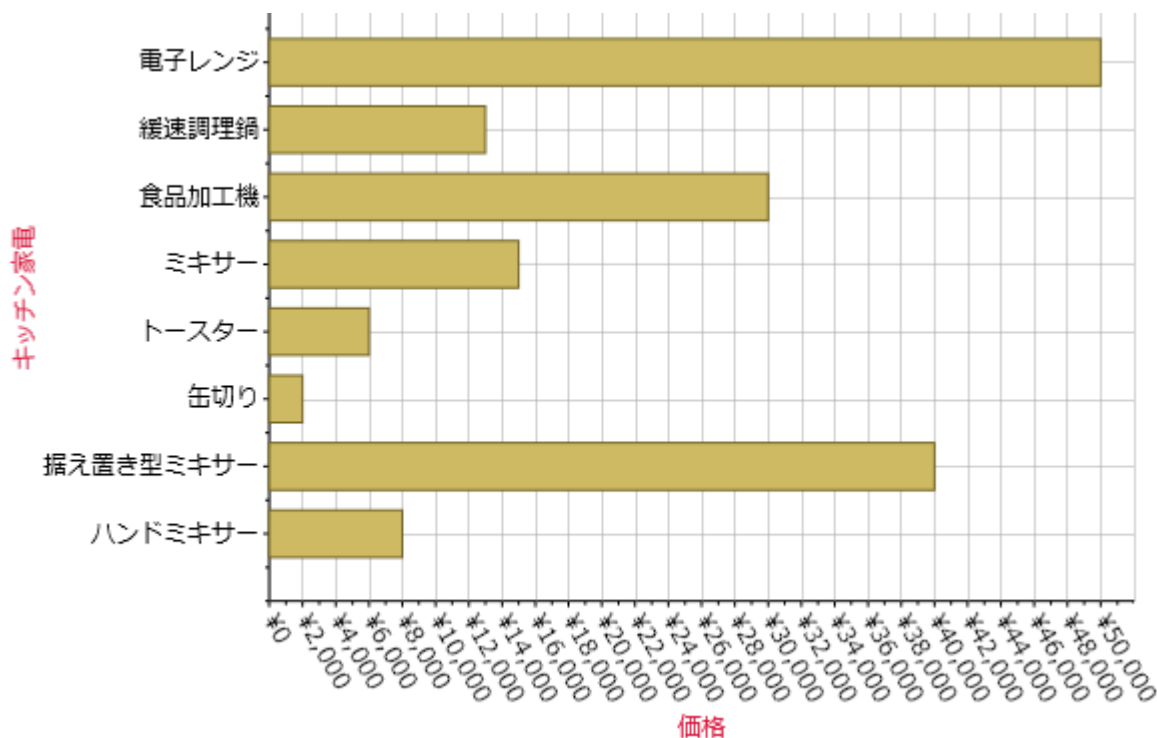
```
c1Chart1.View.AxisX.AnnoAngle=60;
```

Chart for WPF/Silverlight

次の「[手順 4: グラフの外観の調整](#)」では、プログラムでグラフの外観をカスタマイズする方法を学びます。

プログラムを実行して、以下を確認します。

軸の注釈の新しい書式がグラフに適用されます。



次の手順では、**Theme** プロパティのオプションの1つを使用して、グラフの外観をカスタマイズします。

手順 4: グラフの外観の調整

この最後の手順では、**Theme** プロパティを使用してグラフの外観を調整します。

Visual Studio で **XAML** を使用してグラフのテーマを設定するには、以下の作業を実行します。

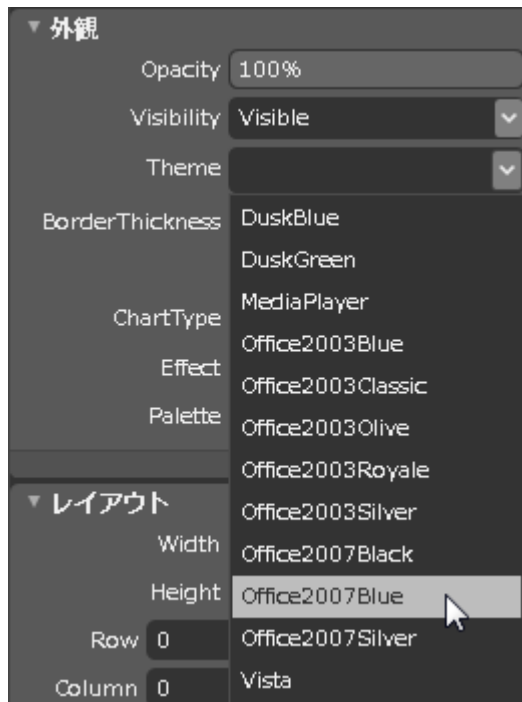
グラフで **Office2007Blue** テーマを定義するには、次のように Theme XAML を `<c1chart:C1Chart>` タグに追加します。

XAML

```
<c1chart:C1Chart Margin="0,0,8,8" MinHeight="160" MinWidth="240" Content="C1Chart"
ChartType="Bar"
Theme="{DynamicResource {ComponentResourceKey TypeInTargetAssembly=c1chart:C1Chart,
ResourceId=Office2007Blue}}">
```

Blend で [プロパティ] ウィンドウを使用してグラフのテーマを設定するには、以下の作業を実行します。

1. **Window1** で **C1Chart** コントロールを選択してアクティブにします。
2. [プロパティ] ウィンドウで、C1Chart の「**外観**」グループに移動します。
3. **Theme** プロパティの横にあるドロップダウン矢印をクリックして、**Office2007Blue** を選択します。



コードビハインドファイルでプログラムによりグラフのテーマを設定するには、以下の作業を実行します。

グラフで **Office2007Blue** テーマを定義するには、次のコードをプロジェクトに追加します。

Visual Basic

```
c1Chart1.Theme = TryCast(c1Chart1.TryFindResource( _
    New ComponentResourceKey(GetType(C1Chart), "Office2007Blue"), _
    ResourceDictionary))
```

C#

```
c1Chart1.Theme = c1Chart1.TryFindResource(
    new ComponentResourceKey(typeof(C1.WPF.C1Chart.C1Chart),
    "Office2007Blue")) as ResourceDictionary;
```

プログラムを実行して、以下を確認します。

Office 2007 Blue テーマが **C1Chart** コントロールに適用されます。

おめでとうございます。**Chart for WPF** クイックスタートを完了して、グラフアプリケーションの作成、グラフへのデータの追加、軸の範囲の設定、軸の注釈の書式設定、およびグラフの外観のカスタマイズが終了しました。

重要なヒント

C1Chart コントロールを使用するときは、以下に示す **Chart for WPF/Silverlight** の重要なヒントを参考にしてください。

ヒント1: BeginUpdate()/EndUpdate メソッドを使用してパフォーマンスを向上させる

チャートのプロパティまたはデータ値を大量に更新する場合は、BeginUpdate()/EndUpdate() ブロック内に更新コードを配置

Chart for WPF/Silverlight

します。

これにより、再描画が**EndUpdate()** メソッドの呼び出し終了後に1回実行されるだけになるので、パフォーマンスが向上します。

次に例を示します。

Visual Basic

・ 更新を開始します

```
C1Chart1.BeginUpdate()
Dim nser As Integer = 10, npts As Integer = 100
For iser As Integer = 1 To nser
    ' データ配列を作成します
    Dim x(npts - 1) As Double, y(npts - 1) As Double
    For ipt As Integer = 0 To npts - 1
        x(ipt) = ipt
        y(ipt) = (1 + 0.05 * iser) * Math.Sin(0.1 * ipt + 0.1 * iser)
    Next
    ' データ系列を作成し、チャートに追加します
    Dim ds = New XYDataSeries()
    ds.XValuesSource = x
    ds.ValuesSource = y
    C1Chart1.Data.Children.Add(ds)
Next
' チャートタイプを設定します
C1Chart1.ChartType = ChartType.Line
' 更新を終了します
C1Chart1.EndUpdate()
```

C#

```
// 更新を開始します
c1Chart1.BeginUpdate();
int nser = 10, npts = 100;
for (int iser = 0; iser < nser; iser++)
{
    // データ配列を作成します
    double[] x = new double[npts], y = new double[npts];
    for (int ipt = 0; ipt < npts; ipt++)
    {
        x[ipt] = ipt;
        y[ipt] = (1 + 0.05 * iser) * Math.Sin(0.1 * ipt + 0.1 * iser);
    }
    // データ系列を作成し、チャートに追加します
    XYDataSeries ds = new XYDataSeries();
    ds.XValuesSource = x; ds.ValuesSource = y;
    c1Chart1.Data.Children.Add(ds);
}
// チャートタイプを設定します
c1Chart1.ChartType = ChartType.Line;
```

```
// 更新を終了します
c1Chart1.EndUpdate();
```

ヒント2: 大きなデータ配列には、チャートタイプとして折れ線グラフまたはエリアグラフを使用する

大量のデータ値がある場合は、折れ線グラフとエリアグラフが最も高いパフォーマンスを提供します。

パフォーマンスをさらに向上するには、**LineAreaOptions.OptimizationRadius**という添付プロパティを設定して、大量のデータに対する組み込み最適化機能を有効にします。次に例を示します。

Visual Basic

```
LineAreaOptions.SetOptimizationRadius(C1Chart1, 1.0)
```

C#

```
LineAreaOptions.SetOptimizationRadius(c1Chart1, 1.0);
```

半径には、1.0 ~ 2.0 の小さな値を使用することをお勧めします。これより大きな値を指定すると、プロットの精度に影響することがあります。

ヒント3: **DataSeries.PlotElementLoaded** イベントを使用してプロット要素の外観と動作を更新する

横棒、縦棒、円などのプロット要素が読み込まれると、**PlotElementLoaded** イベントが発生します。このイベントが発生すると、プロット要素のプロパティおよび対応するデータポイントにアクセスできます。次のコードでは、点の y 値に基づいてその色を設定します。次に例を示します。

Visual Basic

データ配列を作成します

```
Dim npts As Integer = 100
Dim x(npts - 1) As Double, y(npts - 1) As Double
For ipt As Integer = 0 To npts - 1
    x(ipt) = ipt
    y(ipt) = Math.Sin(0.1 * ipt)
Next
```

データ系列を作成します

```
Dim ds = New XYDataSeries()
ds.XValuesSource = x
ds.ValuesSource = y
```

イベントハンドラを設定します

```
AddHandler ds.PlotElementLoaded, AddressOf PlotElement_Loaded
```

データ系列をチャートに追加します

```
C1Chart1.Data.Children.Add(ds)
```

チャートタイプを設定します

```
C1Chart1.ChartType = ChartType.LineSymbols
```

...

イベントハンドラ

```
Sub PlotElement_Loaded(ByVal sender As Object, ByVal args As EventArgs)
    Dim pe = CType(sender, PlotElement)
```

Chart for WPF/Silverlight

```
If Not TypeOf pe Is Lines Then
    Dim dp As DataPoint = pe.DataPoint
    ' 正規化された y 値(0~1)
    Dim nval As Double = 0.5 * (dp.Value + 1)
    ' 青(-1)から赤(+1)で塗りつぶします
    pe.Fill = New SolidColorBrush(Color.FromRgb(CByte(255 * nval), _
        0, CByte(255 * (1 - nval))))
End If
End Sub
```

C#

```
// データ配列を作成します
int npts = 100;
double[] x = new double[npts], y = new double[npts];
for (int ipt = 0; ipt < npts; ipt++)
{
    x[ipt] = ipt;
    y[ipt] = Math.Sin(0.1 * ipt);
}
// データ系列を作成します
XYDataSeries ds = new XYDataSeries();
ds.XValuesSource = x; ds.ValuesSource = y;
// イベントハンドラを設定します
ds.PlotElementLoaded += (s, e) =>
{
    PlotElement pe = (PlotElement)s;
    if (!(pe is Lines)) // 線をスキップします
    {
        DataPoint dp = pe.DataPoint;
        // 正規化された y 値(0~1)
        double nval = 0.5*(dp.Value + 1);
        // 青(-1)から赤(+1)で塗りつぶします
        pe.Fill = new SolidColorBrush(
            Color.FromRgb((byte)(255 * nval), 0, (byte)(255 * (1-nval))));
    }
};
// データ系列をチャートに追加します
c1Chart1.Data.Children.Add(ds);
// チャートタイプを設定します
c1Chart1.ChartType = ChartType.LineSymbols;
```

ヒント 4: データポイントのラベルとツールチップ

データポイントのラベルまたはツールチップを作成するには、**PointLabelTemplate** プロパティまたは **PointTooltipTemplate** プロパティにデータテンプレートを設定する必要があります。

次のサンプルコードは、各データポイントのインデックスを表示する方法を示します。

XAML

```
<c1chart:C1Chart Name="c1Chart1" ChartType="XYPlot">
    <c1chart:C1Chart.Data>
```



```

<clchart:ChartData>
  <!-- source collection -->
  <clchart:ChartData.ItemsSource>
    <PointCollection>
      <Point X="1" Y="1" />
      <Point X="2" Y="2" />
      <Point X="3" Y="3" />
      <Point X="4" Y="2" />
      <Point X="5" Y="1" />
    </PointCollection>
  </clchart:ChartData.ItemsSource>
  <clchart:XYDataSeries SymbolSize="16,16"
    XValueBinding="{Binding X}" ValueBinding="{Binding Y}">
    <clchart:XYDataSeries.PointLabelTemplate>
      <DataTemplate>
        <!-- display point index at the center of point symbol -->
        <TextBlock clchart:PlotElement.LabelAlignment="MiddleCenter"
          Text="{Binding PointIndex}" />
      </DataTemplate>
    </clchart:XYDataSeries.PointLabelTemplate>
  </clchart:XYDataSeries>
</clchart:ChartData>
</clchart:ClChart.Data>
</clchart:ClChart>

```

テンプレートから作成される要素のデータコンテキストは、対応するデータポイントに関する情報を含む `DataPoint` クラスのインスタンスに設定されます。

ヒント5: チャートを画像として保存する

次のメソッドは、チャート画像を png ファイルとして保存します。

Visual Basic

```

Sub Using stm = System.IO.File.Create(fileName)
  clChart1.SaveImage(stm, ImageFormat.Png)
End Using

```

C#

```

using (var stm = System.IO.File.Create(fileName))
{
  clChart1.SaveImage(stm, ImageFormat.Png);
}

```

ヒント6: チャートの印刷

次のコードは、デフォルト設定を使用して、指定されたチャートをデフォルトのプリンタに印刷します。次に例を示します。

WPF

Chart for WPF/Silverlight

```
Dim pd = New PrintDialog()  
pd.PrintVisual(C1Chart1, "chart")
```

```
new PrintDialog().PrintVisual(c1Chart1, "chart");
```

Silverlight

Visual Basic

```
Imports System.Windows.Printing  
Dim doc As New PrintDocument  
AddHandler doc.PrintPage, _  
Sub(s, ea)  
    ea.PageVisual = C1Chart1  
End Sub  
doc.Print("チャート印刷")
```

C#

```
using System.Windows.Printing;  
PrintDocument doc = new PrintDocument();  
doc.PrintPage += (s, ea) =>  
{  
    ea.PageVisual = c1Chart1;  
};  
doc.Print("チャート印刷");
```

ヒント7: デカルト座標の複数のチャートタイプを混在させる

ChartType プロパティを使用すると、同じデカルト座標プロットに異なるチャートタイプを簡単に混在させることができます。

次のコードは、3つのデータ系列を作成する方法を示します。1つ目は面、2つ目は階段、3つ目はデフォルトチャートタイプ(折れ線)を含む系列です。

Visual Basic

```
Dim nser As Integer = 3, npts As Integer = 25  
For iser As Integer = 1 To nser  
    ' データ配列を作成します  
    Dim x(npts - 1) As Double, y(npts - 1) As Double  
    For ipt As Integer = 0 To npts - 1  
        x(ipt) = ipt  
        y(ipt) = (1 + 0.05 * iser) * Math.Sin(0.1 * ipt + 0.1 * iser)  
    Next  
    ' データ系列を作成し、チャートに追加します  
    Dim ds = New XYDataSeries()  
    ds.XValuesSource = x  
    ds.ValuesSource = y  
    C1Chart1.Data.Children.Add(ds)
```

Next

' デフォルトのチャートタイプ

```
C1Chart1.ChartType = ChartType.Line
```

' 1つ目の系列

```
C1Chart1.Data.Children(0).ChartType = ChartType.Area
```

' 2つ目の系列

```
C1Chart1.Data.Children(1).ChartType = ChartType.Step
```

C#

```
int nser = 3, npts = 25;
for (int iser = 0; iser < nser; iser++)
{
    // データ配列を作成します
    double[] x = new double[npts], y = new double[npts];
    for (int ipt = 0; ipt < npts; ipt++)
    {
        x[ipt] = ipt;
        y[ipt] = (1 + 0.05 * iser) * Math.Sin(0.1 * ipt + 0.1 * iser);
    }
    // データ系列を作成し、チャートに追加します
    XYDataSeries ds = new XYDataSeries();
    ds.XValuesSource = x; ds.ValuesSource = y;
    c1Chart1.Data.Children.Add(ds);
}
//デフォルトのチャートタイプ
c1Chart1.ChartType = ChartType.Line;
// 1つ目の系列
c1Chart1.Data.Children[0].ChartType = ChartType.Area;
// 2つ目の系列
c1Chart1.Data.Children[1].ChartType = ChartType.Step;
```

XAML クイックリファレンス

次の XAML は、チャートタイプの選択、パレットの選択、軸の設定、およびデータ系列の追加を行う方法を示します。

XAML

```
<c1:C1Chart x:Name="_chart" Palette="Module" ChartType="Line"
Foreground="#a0000000" Background="#e0ffffff" >

    <c1:C1Chart.View>
        <c1:ChartView>
            <c1:ChartView.AxisX>
                <c1:Axis Title="年" MajorGridStroke="Transparent" />
            </c1:ChartView.AxisX>
            <c1:ChartView.AxisY>
                <c1:Axis Title="四半期売上高(単位:\10,00,000)"
MajorGridStroke="#40000000" AnnoFormat="n0" />
            </c1:ChartView.AxisY>
        </c1:ChartView>
    </c1:C1Chart.View>
</c1:C1Chart>
```

Chart for WPF/Silverlight

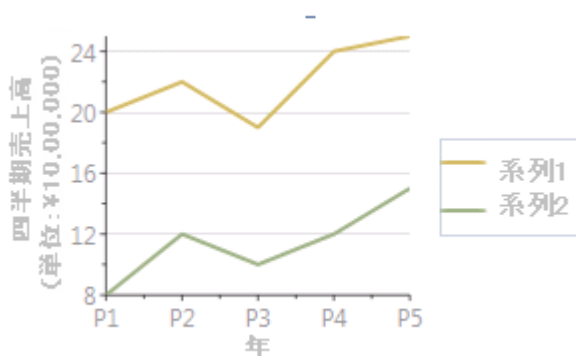
```
</cl:C1Chart.View>

<cl:C1ChartLegend Position="Right" />

<cl:C1Chart.Data>
  <cl:ChartData ItemNames="P1 P2 P3 P4 P5">
    <cl:DataSeries Label="s1" Values="20, 22, 19, 24, 25"
ConnectionStrokeThickness="6" />
    <cl:DataSeries Label="s2" Values="8, 12, 10, 12, 15" />
  </cl:ChartData>
</cl:C1Chart.Data>
</cl:C1Chart>
```

例: 基本的な折れ線グラフを設定する

次の XAML は、C1Chart コントロールを宣言し、グラフの基本的な外観を定義するには、ChartType プロパティ、Theme プロパティおよび Palette プロパティを設定する方法を示します。XAML コードをタグ内<Grid>...</Grid>に追加します。以下のグラフは、次の XAML コードを使用して生成されます。



XAML

```
<Window x:Class="WpfApplication1.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:cl="http://schemas.componentone.com/xaml/clchart"
  Title="Window1" Height="300" Width="348" >
  <Grid>
    <!--C1Chart コントロールを宣言します。
    グラフの基本的な外観を定義するには、
    ChartType プロパティ、Theme プロパティおよび Palette プロパティを設定します。-->
    <cl:C1Chart
      Name="C1Chart1"
      ChartType="Line"
      Foreground="#a0000000"
      Background="#e0ffffff"
      Theme ="Vista"
      Palette ="Aspect" >
      <!-- グラフの軸を含むグラフビューを定義します。-->
      <cl:C1Chart.View>
        <cl:ChartView>
          <!-- X 軸を定義します。(タイトル、グリッド) -->
```

```

<cl:ChartView.AxisX>
  <cl:Axis
    Title="年"
    MajorGridStroke="Transparent"/>
</cl:ChartView.AxisX>
<!-- Y 軸を定義します。(タイトル、グリッド、注釈書式) -->
<cl:ChartView.AxisY>
  <cl:Axis
    Title="四半期売上高(単位:\10,00,000)"
    MajorGridStroke="#40000000"
    AnnoFormat="n0" />
</cl:ChartView.AxisY>
</cl:ChartView>
</cl:C1Chart.View>
<!-- データ系列を含むグラフデータを定義します。-->
<cl:C1Chart.Data>
  <cl:ChartData>
    <!-- ItemNames は、X 軸上のラベルを定義します。 -->
    <cl:ChartData.ItemNames>P1 P2 P3 P4 P5</cl:ChartData.ItemNames>
    <!-- 各データ系列は、ラベル(凡例で表示される)
    および系列のデータを指定します。-->
    <cl:DataSeries Label="Series 1" RenderMode="Default" Values="20 22 19 24
25" />
    <cl:DataSeries Label="Series 2" RenderMode="Default" Values="8 12 10 12 15"
/>
  </cl:ChartData>
</cl:C1Chart.Data>
<!-- 系列とそのスタイルを含む凡例を表示するには、
グラフの右側にドッキングされている ChartLegend を追加します。-->
<cl:C1ChartLegend DockPanel.Dock="Right" />
</cl:C1Chart>
</Grid>
</Window>

```

例:ガントチャートを設定する

ガントチャートを作成するには、次の XAML コードを使用します。

XAML

```

<clchart:C1Chart Margin="0" Name="c1Chart1"
  xmlns:sys="clr-namespace:System;assembly=microsoft.windows.common-core-1.0" >
  <clchart:C1Chart.Resources>
    <x:Array x:Key="start" Type="sys:DateTime" >
      <sys:DateTime>2008-6-1</sys:DateTime>
      <sys:DateTime>2008-6-4</sys:DateTime>
      <sys:DateTime>2008/06/02</sys:DateTime>
    </x:Array>
    <x:Array x:Key="end" Type="sys:DateTime">
      <sys:DateTime>2008/06/10</sys:DateTime>
      <sys:DateTime>2008/06/12</sys:DateTime>
      <sys:DateTime>2008/06/15</sys:DateTime>
    </x:Array>
  </clchart:C1Chart.Resources>
</clchart:C1Chart>

```

Chart for WPF/Silverlight

```
</x:Array>
</clchart:C1Chart.Resources>
<clchart:C1Chart.Data>
  <clchart:ChartData>
    <clchart:ChartData.Renderer>
      <clchart:Renderer2D Inverted="True" ColorScheme="Point"/>
    </clchart:ChartData.Renderer>
    <clchart:ChartData.ItemNames>Task1 Task2 Task3</clchart:ChartData.ItemNames>
    <clchart:HighLowSeries HighValuesSource="{StaticResource end}"
      LowValuesSource="{StaticResource start}"/>
  </clchart:ChartData>
</clchart:C1Chart.Data>
<clchart:C1Chart.View>
  <clchart:ChartView>
    <clchart:ChartView.AxisX>
      <clchart:Axis IsTime="True" AnnoFormat="d"/>
    </clchart:ChartView.AxisX>
  </clchart:ChartView>
</clchart:C1Chart.View>
</clchart:C1Chart>
```

例: 積層面グラフを作成する

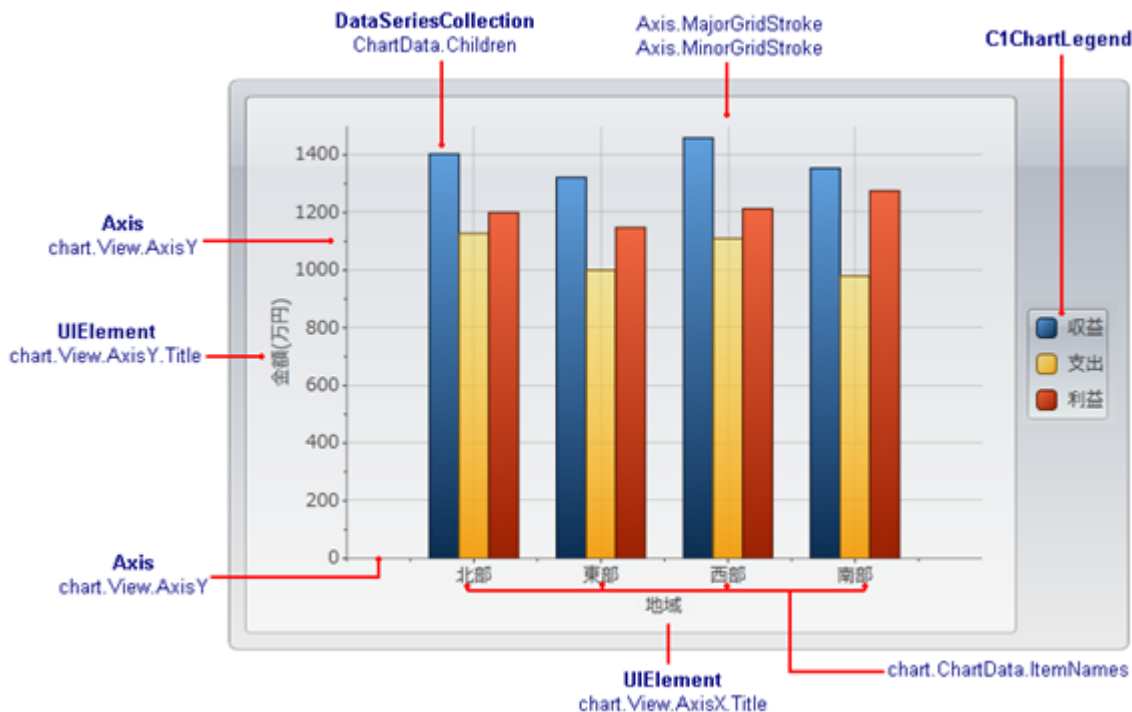
積層面グラフを作成するには、次の XAML コードのように `DataSeries.ChartType` の代わりに `C1Chart.ChartType` プロパティを設定する必要があります。

XAML

```
<cl:C1Chart ChartType="AreaStacked" >
  <cl:C1Chart.Data>
    <cl:ChartData ItemNames="P1 P2 P3 P4 P5">
      <cl:DataSeries Label="Series 1" Values="20 22 19 24 25" />
      <cl:DataSeries Label="Series 2" Values="8 12 10 12 15" />
    </cl:ChartData>
  </cl:C1Chart.Data>
  <cl:C1ChartLegend DockPanel.Dock="Right" />
</cl:C1Chart>
```

グラフの要素

C1Chart コントロールを使用してグラフを作成し、書式設定する場合、主なプロパティとグラフ要素の対応関係を理解していると有益です。下の図は、これを示しています。



標準的なグラフの作成に関わる手順は、以下のとおりです。

1. グラフタイプの選択 (**ChartType**プロパティ)

C1Chart では、横棒、縦棒、折れ線、面、円、ラジアル、極、ローソクなど、約 30 のグラフタイプがサポートされています。最適なグラフのタイプは、主にデータの性質によって決まります。これについては後で説明します。

2. 軸の設定 (**AxisX**プロパティと**chart.View.AxisY**プロパティ)

軸の設定では、軸のタイトル、目盛記号の主間隔と副間隔、目盛記号の横に表示されるラベルの内容と書式を指定するのが普通です。

3. 1つ以上のデータ系列の追加 (**chart.Data.Children**コレクション)

この手順では、グラフ上の系列ごとに1つの **DataSeries** オブジェクトを作成してデータを入力し、そのオブジェクトを **chart.Data.Children** コレクションに追加します。データにおいて各ポイントに数値が1つ (Y 座標) しか含まれない場合は、通常の **DataSeries** オブジェクトを使用します。データにおいて各ポイントに数値が2つ (X 座標と Y 座標) 含まれる場合は、**XYDataSeries** オブジェクトを使用します。

4. **Theme**プロパティと**Palette**プロパティを使用したグラフの外観の調整

Theme プロパティでは、グラフ全体の外観を制御する 10 種類を超える組み込みプロパティセットの1つを選択できます。**Palette** プロパティでは、データ系列の色の指定に使用する 20 種類を超える組み込み色パレットの1つを選択できます。これら2つのプロパティを組み合わせれば、実用レベルの外観を持ったグラフをわずかな作業で作成できる、約 200 種類ものオプションが提供されます。

グラフ種別

組み込みタイプを使用すると、グラフの外観を最も簡単に設定できます。たとえば、積み上げ横棒グラフを設定する場合は、それに対応する文字列を**ChartType** プロパティに指定するだけです。

XAML

```
<c1c:C1Chart ChartType="Bar.Stacked">
    ...
</c1c:C1Chart>
```

次の表に、使用可能な組み込みグラフタイプを一覧します。実行時には、**ChartTypes.GetAvailableChartTypes()** メソッドを使用することで、すべての組み込みグラフタイプのリストを取得できます。

表 1 組み込みグラフタイプ

| ギャラリーでの名前 |
|-------------------------|
| Area |
| AreaSmoothed |
| AreaStacked |
| AreaStacked100pc |
| Bar |
| BarStacked |
| BarStacked100pc |
| Bubble |
| Candle |
| Column |
| ColumnStacked |
| ColumnStacked100pc |
| Gantt |
| HighLowOpenClose |
| Line |
| LineSmoothed |
| LineStacked |
| LineStacked100pc |
| LineSymbols |
| LineSymbolsSmoothed |
| LineSymbolsStacked |
| LineSymbolsStacked100pc |
| Pie |

| |
|--------------------------------|
| PieDoughnut |
| PieExploded |
| PieExplodedDoughnut |
| PieStacked |
| PolarLines |
| PolarLinesSymbols |
| PolarSymbols |
| Polygon |
| Radar |
| RadarFilled |
| RadarSymbols |
| Step |
| StepArea |
| StepSymbols |
| XYPlot |
| Area3D (WPF のみ) |
| Area3DSmoothed (WPF のみ) |
| Area3DStacked (WPF のみ) |
| Area3DStacked100pc (WPF のみ) |
| Bar3D (WPF のみ) |
| Bar3DStacked (WPF のみ) |
| Bar3DStacked100pc (WPF のみ) |
| Pie3D (WPF のみ) |
| Pie3DDoughnut (WPF のみ) |
| Pie3DExploded (WPF のみ) |
| Pie3DExplodedDoughnut (WPF のみ) |
| Ribbon (WPF のみ) |

エリアグラフ

エリアグラフでは、相互接続されたデータポイントとして各系列が表示され、それらのポイントの下の領域は塗りつぶされます。各系列は、前の系列の上に表示されます。各系列は単独で表示したり、積み重ねて表示したりできます。**Chart for WPF/Silverlight** では、以下のタイプの**エリア**グラフがサポートされています。

- Area3D (WPF のみ)
- Area3DSmoothed (WPF のみ)
- Area3DStacked (WPF のみ)
- Area3DStacked100pc (WPF のみ)

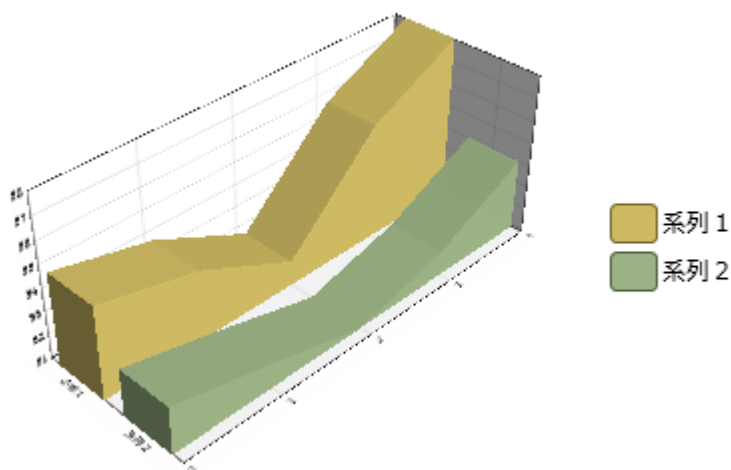
Chart for WPF/Silverlight

- AreaSmoothed
- AreaStacked
- AreaStacked100pc

3D エリアグラフ (WPF のみ)

AreaShape3D クラスは、**3D エリアグラフ**のプロット要素に関連付けられたデータポイントにアクセスする、マウスポインタがプロット要素上に置かれているときにその要素の値を取得する、プロット要素のサイズ(ピクセル)を取得または設定する、各ポイントを滑らかな線で接続するかどうかを指定するといった場合に使用します。

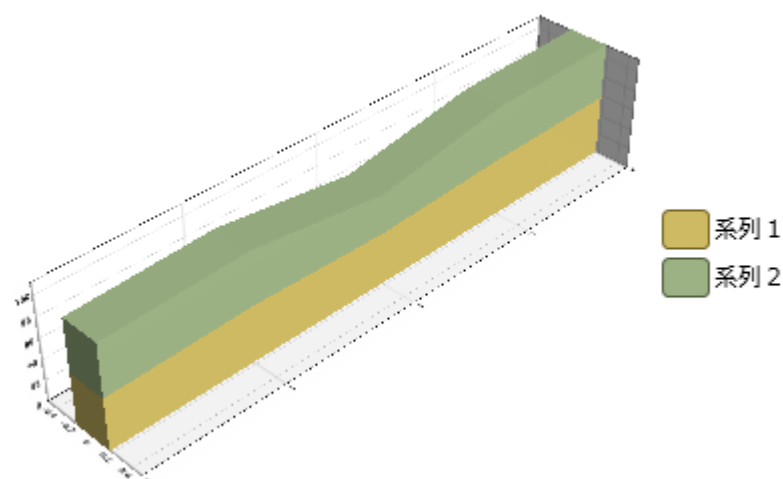
次の図は、**ChartType** プロパティを **Area3D** に設定したときの **3D エリアグラフ**を表します。



積み重ね 3D エリアグラフ (WPF のみ)

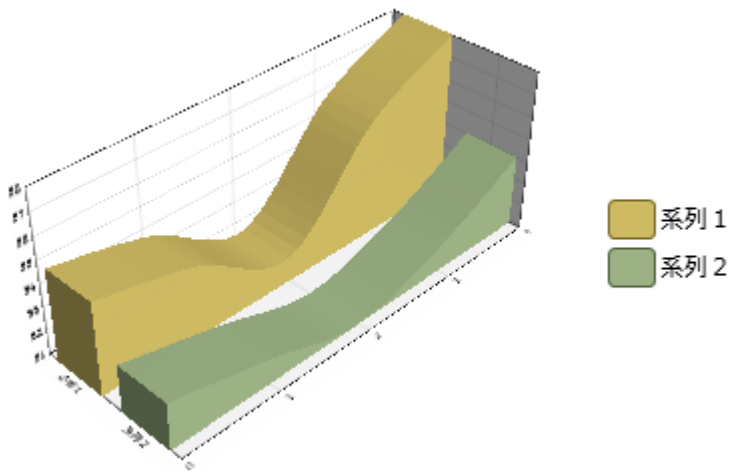
BaseRenderer の **Stacked** プロパティを使用し、**StackedOptions** 列挙を設定して、**Stacked** または **Stacked 100%** など、特定の積み重ねエリアグラフを作成します。積み重ねグラフは、各系列の値を前の系列の値の上に積み重ねることによってデータを表します。

次の図は、**ChartType** プロパティを **Area3Dstacked** に設定したときの **積み重ね 3D エリアグラフ**を表します。



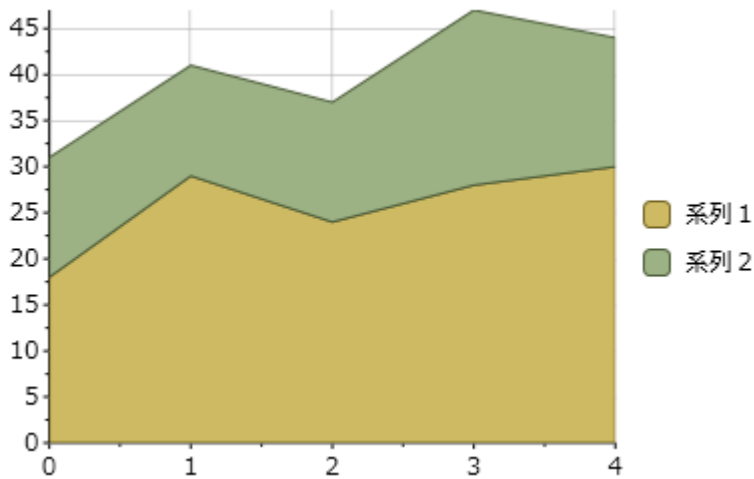
3D 平滑化エリアグラフ (WPF のみ)

次の図は、**ChartType** プロパティを **Area3Dsmoothed** に設定したときの **平滑化エリアグラフ**を表します。



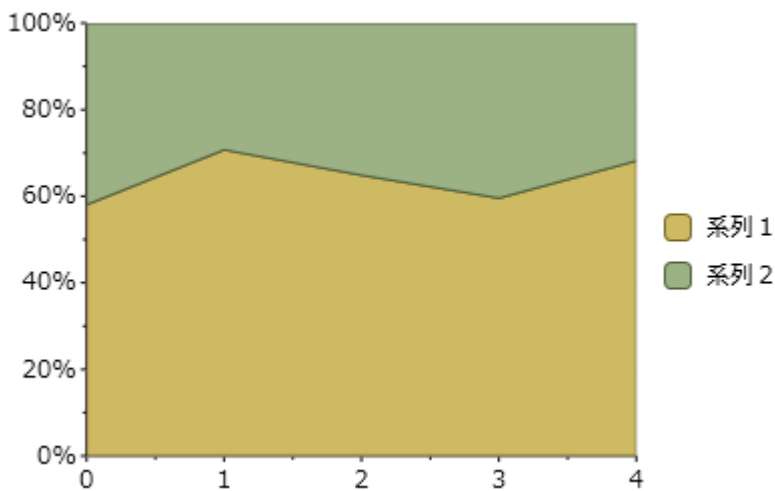
積み重ねエリアグラフ

次の図は、**ChartType** プロパティを **AreaStacked** に設定したときの**積み重ねエリアグラフ**を表します。



100% 積み重ねエリアグラフ

次の図は、**ChartType** プロパティを **AreaStacked100pc** に設定したときの**100% 積み重ねエリアグラフ**を表します。



エリアグラフを作成するには、次のマークアップを使用します。

Chart for WPF/Silverlight

XAML

```
<cl:C1Chart ChartType="Area" >
  <cl:C1Chart.Data>
    <cl:ChartData ItemNames="P1 P2 P3 P4 P5">
      <cl:DataSeries Label="Series 1" Values="20 22 19 24 25" />
      <cl:DataSeries Label="Series 2" Values="8 12 10 12 15" />
    </cl:ChartData>
  </cl:C1Chart.Data>
  <cl:C1ChartLegend DockPanel.Dock="Right" />
</cl:C1Chart>
```

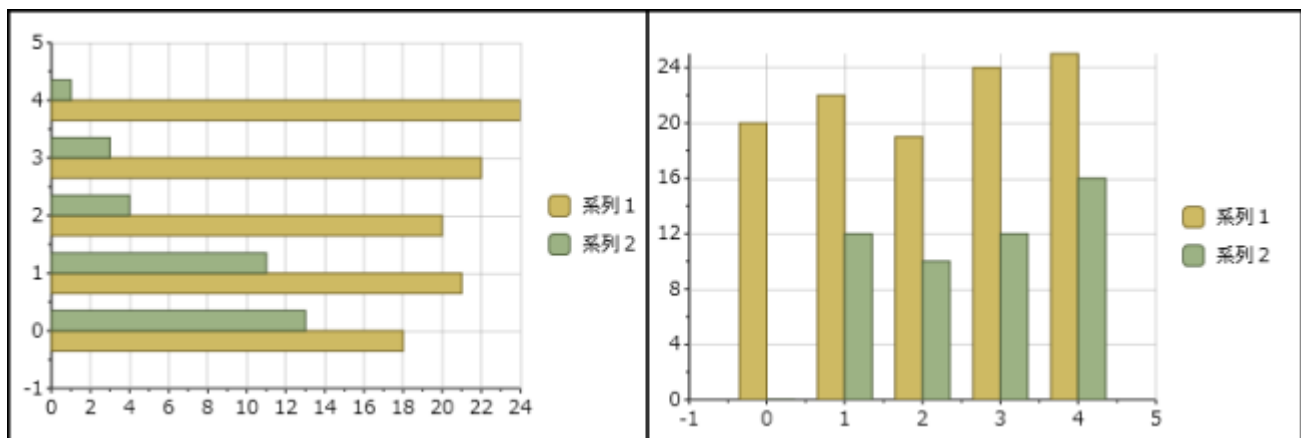
横棒グラフと縦棒グラフ

Chart for WPF/Silverlight では、以下のタイプの横棒グラフと縦棒グラフがサポートされています。

- Bar または Column
- Bar3D または Column3D
- Bar3Dstacked または Column3Dstacked
- Bar3Dstacked100pc または Column3Dstacked100pc
- BarStacked または ColumnStacked
- BarStacked100pc または ColumnStacked100pc

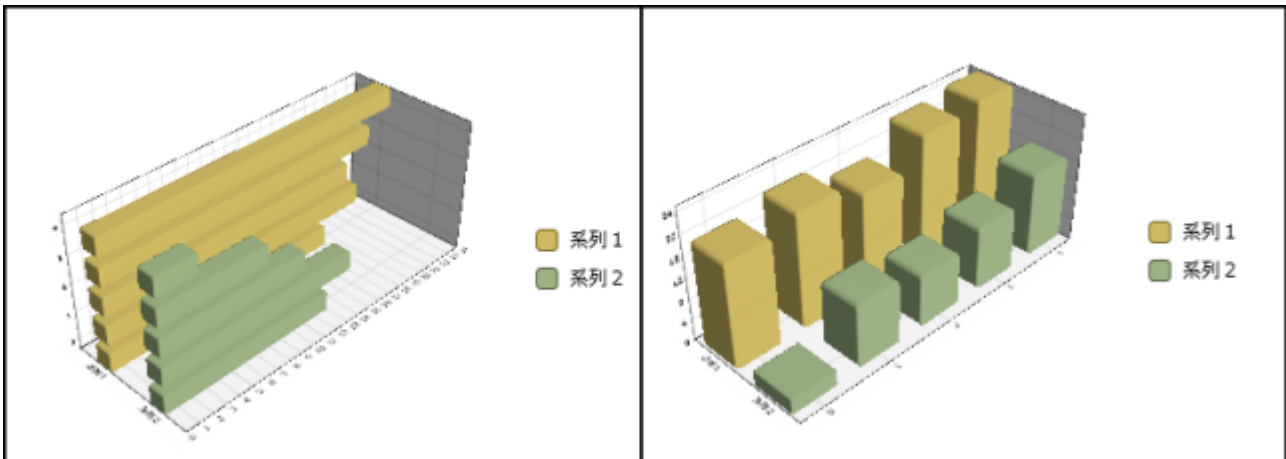
横棒 または 縦棒グラフ

次の図は、ChartType プロパティを Bar または Column に設定したときの横棒グラフまたは縦棒グラフを表します。



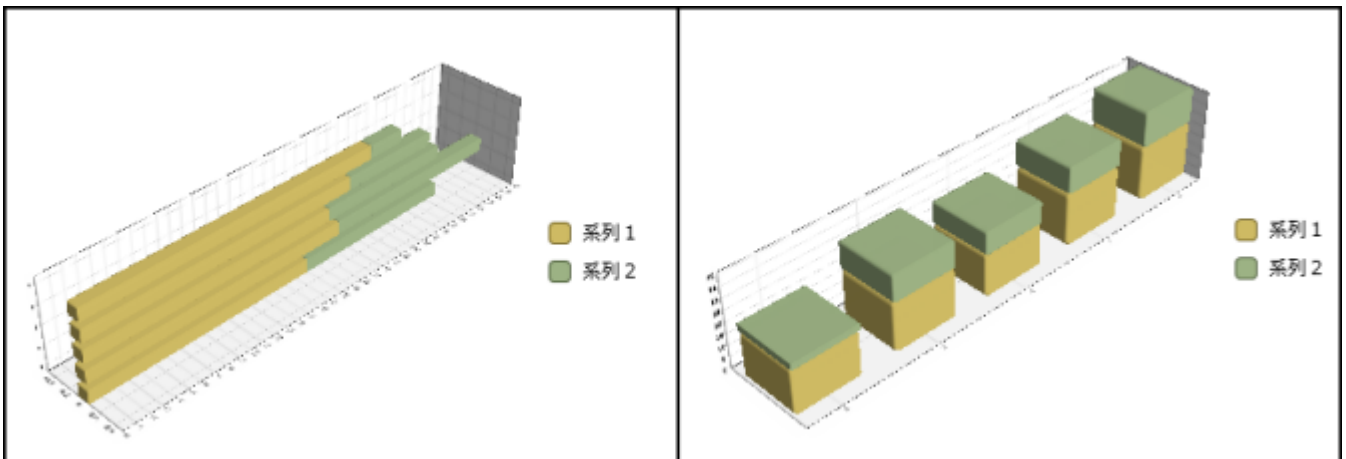
3D 横棒 または 3D 縦棒グラフ (WPF のみ)

次の図は、ChartType プロパティを Bar3D または Column3D に設定したときの 3D 横棒グラフまたは 3D 縦棒グラフを表します。



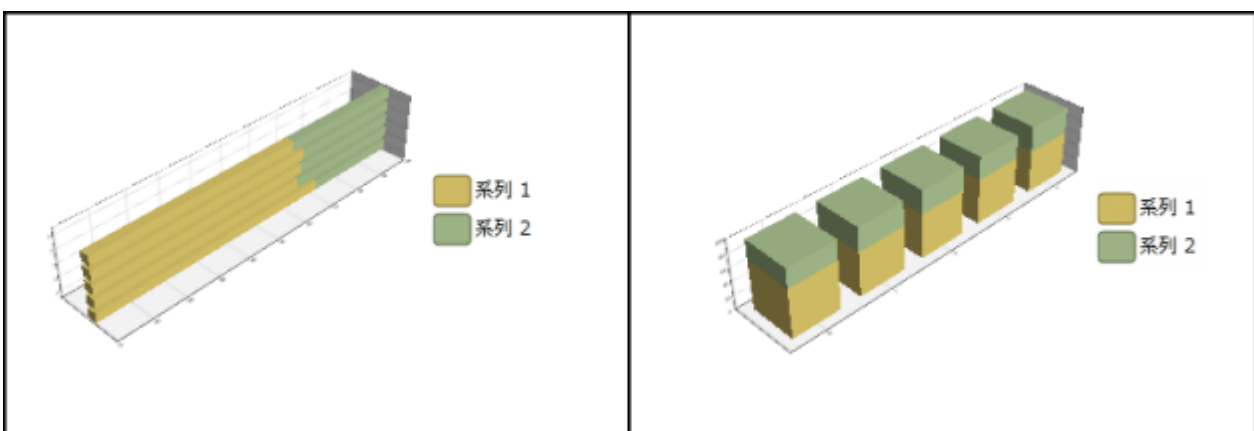
積み重ね3D横棒 または 積み重ね3D縦棒グラフ (WPF のみ)

次の図は、**ChartType** プロパティを **Bar3DStacked** または **Column3DStacked** に設定したときの積み重ね 3D 横棒グラフまたは積み重ね 3D 縦棒グラフを表します。



100%積み重ね3D横棒 または 100%積み重ね3D縦棒グラフ (WPF のみ)

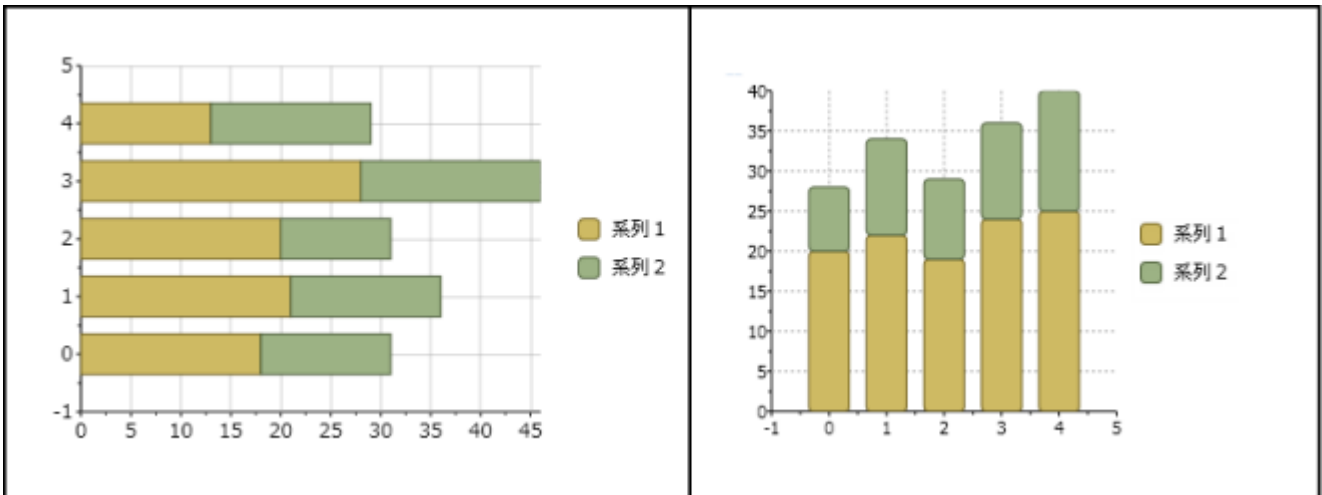
次の図は、**ChartType** プロパティを **Bar3DStacked100pc** または **Column3DStacked100pc** に設定したときの 100% 積み重ね 3D 横棒グラフまたは100% 積み重ね 3D 縦棒グラフを表します。



積み重ね横棒 または 積み重ね縦棒グラフ

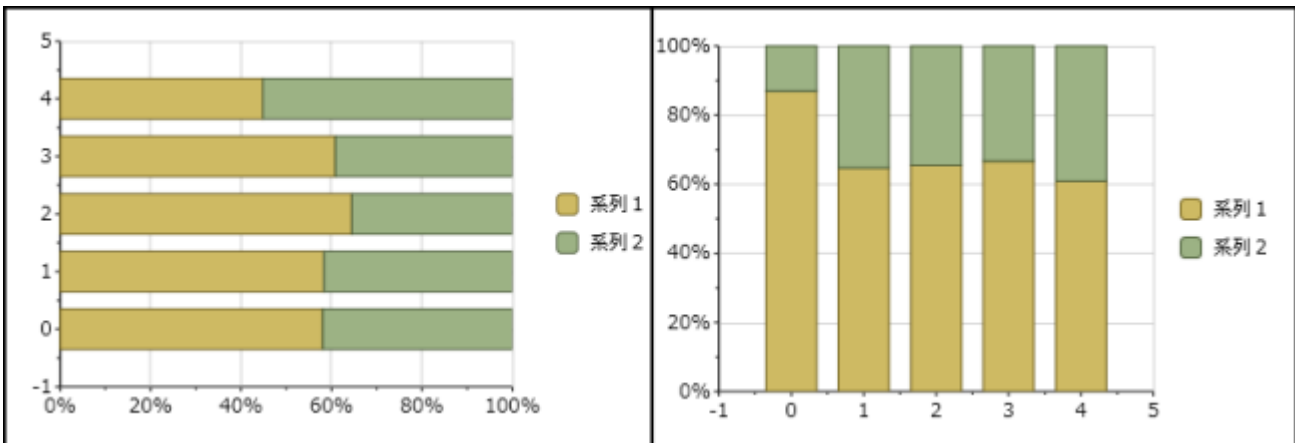
次の図は、**ChartType** プロパティを **BarStacked** または **ColumnStacked** に設定したときの積み重ね横棒グラフまたは積み重ね縦棒グラフを表します。

Chart for WPF/Silverlight



100%積み重ね横棒 または 100%積み重ね縦棒グラフ

次の図は、**ChartType** プロパティを **BarStacked100pc** または **ColumnStacked100pc** に設定したときの **100% 積み重ね横棒グラフ**または**100%積み重ね**グラフを表します。



横棒/縦棒グラフの四角形の角を変更する

デフォルトでは、横棒/縦棒グラフの角は丸くなっていません。四角形の角の半径は、**Bar** クラスを使用して設定できます。次に例を示します。

C#

```
ds.Symbol = new Bar() { RadiusX=5, RadiusY=5};
```

縦棒グラフのマウスクリックイベントの作成

次の XAML コードのように、**MouseDown** イベントや **MouseLeave** イベントを使用して、縦棒グラフの棒がクリックされたときにアニメーションを追加できます。

XAML

```
<Window x:Class="MouseEvent.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:sys="clr-namespace:System;assembly=mscorlib"
```

```

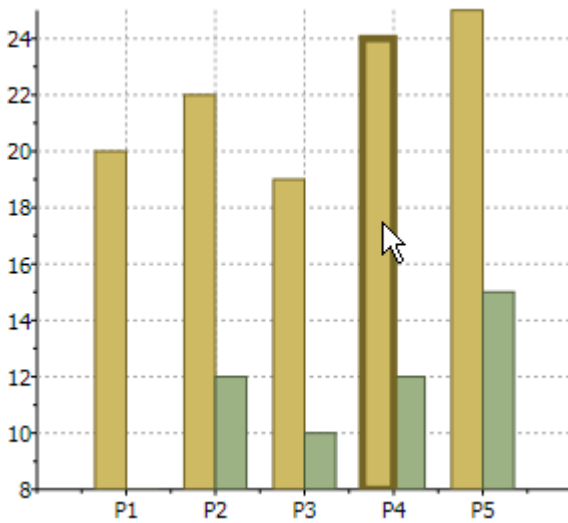
Title="Window1" Height="300" Width="300"
xmlns:clchart="http://schemas.componentone.com/xaml/clchart" Loaded="Window_Loaded">
  <Grid>
    <Grid.Resources>
      <Style x:Key="sstyle" TargetType="{x:Type clchart:PlotElement}">
        <Setter Property="StrokeThickness" Value="1" />
        <Setter Property="Canvas.ZIndex" Value="0" />
        <Style.Triggers>
          <EventTrigger RoutedEvent="clchart:PlotElement.MouseDown">
            <BeginStoryboard>
              <Storyboard>
                <Int32Animation Storyboard.TargetProperty="
(Panel.ZIndex) "
                    To="1" />
                <DoubleAnimation
Storyboard.TargetProperty="StrokeThickness"
                    To="4" Duration="0:0:0.3"
                    AutoReverse="True"
                    RepeatBehavior="Forever" />
              </Storyboard>
            </BeginStoryboard>
          </EventTrigger>
          <EventTrigger RoutedEvent="clchart:PlotElement.MouseLeave">
            <BeginStoryboard>
              <Storyboard>
                <DoubleAnimation
Storyboard.TargetProperty="StrokeThickness" />
                <Int32Animation Storyboard.TargetProperty="
(Panel.ZIndex) " />
              </Storyboard>
            </BeginStoryboard>
          </EventTrigger>
        </Style.Triggers>
      </Style>
    </Grid.Resources>
    <clchart:C1Chart Margin="0" Name="c1Chart1" ChartType="Column">
      <clchart:C1Chart.Data>
        <clchart:ChartData>
          <clchart:ChartData.ItemNames>P1 P2 P3 P4
P5</clchart:ChartData.ItemNames>
          <clchart:DataSeries SymbolStyle="{StaticResource sstyle}"
Values="20
22 19 24 25" />
          <clchart:DataSeries SymbolStyle="{StaticResource sstyle}"
Values="8
12 10 12 15" />
        </clchart:ChartData>
      </clchart:C1Chart.Data>
    </clchart:C1Chart>
  </Grid>
</Window>

```

Chart for WPF/Silverlight

このトピックの作業結果

いずれかの縦棒をクリックすると、四角形の境界線にアニメーションが表示されます。



データ系列の各横棒/縦棒の色を指定する

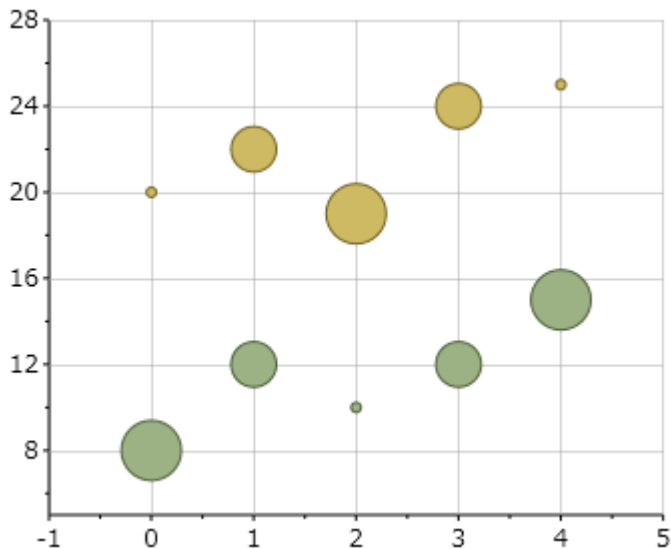
次のコードを使用して、データ系列の **DataSeries.PlotElementLoaded** イベントで各横棒/縦棒の色を指定できます。

XAML

```
var palette = new Brush[] { Brushes.Red, Brushes.Plum, Brushes.Purple };
dataSeries.PlotElementLoaded += (s, e) =>
{
    PlotElement pe = (PlotElement)s;
    if (pe.DataPoint.PointIndex >= 0)
        pe.Fill = palette[pe.DataPoint.PointIndex % palette.Length];
};
```

バブルグラフ

次の図は、**ChartType** プロパティを **Bubble** に設定したときの**バブルグラフ**を表します。



次の XAML コードは、バブルチャートを作成します。

XAML

```
<clchart:C1Chart ChartType="Bubble"
    clchart:BubbleOptions.MinSize="5,5"
    clchart:BubbleOptions.MaxSize="30,30"
    clchart:BubbleOptions.Scale="Area">
  <clchart:C1Chart.Data>
  <clchart:ChartData>
    <clchart:BubbleSeries Values="20 22 19 24 25" SizeValues="1 2 3 2 1" />
    <clchart:BubbleSeries Values="8 12 10 12 15" SizeValues="3 2 1 2 3"/>
  </clchart:ChartData>
  </clchart:C1Chart.Data>
</clchart:C1Chart>
```

ローソク足チャート

ローソク足チャートは、特殊なタイプの Hi-Lo-Open-Close チャートで、始値と終値の関係、および高値と安値の関係を示す際に使用されます。ローソク足チャートは、Hi-Lo-Open-Close チャートと同じ価格データ(時間、高値、安値、始値、終値)を使用しますが、ローソクのように太くなっている部分があることが異なります。

ローソク足チャートは、次の要素で構成されます。

- **ローソク足**

ローソクのように太くなっている部分の色とサイズを使用して、始値と終値に関する追加情報を表します。長い白抜きのローソクは買い圧力を示し、長い塗りつぶされたローソクは売り圧力を示します。

白抜きのローソクは、株価が上昇したこと(終値が始値より高い)を示しています。白抜きのローソクでは、胴体部分の下端が始値を示し、上端が終値を示します。

塗りつぶされたローソクは、株価が下降したこと(始値が終値より高い)ことを示しています。塗りつぶされたローソクでは、胴体部分の上端が始値を示し、下端が終値を示します。

- **上ヒゲ**

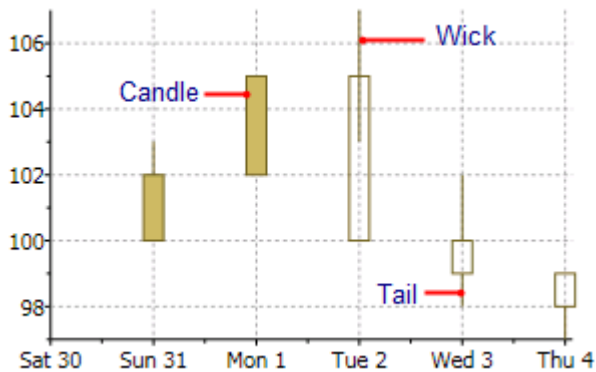
上ヒゲは、株価の高値を表現するローソクの上にある線です。

- **下ヒゲ**

Chart for WPF/Silverlight

下ヒゲは、株価の安値を表現するローソクの下にある線です。

以下の図では、ローソク足チャートの要素に名前を付けました。

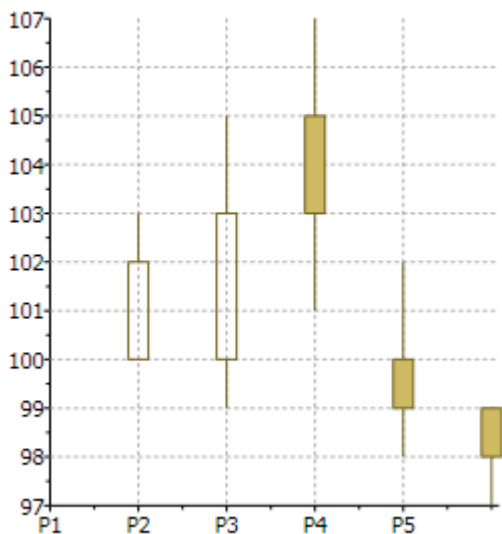


次の図は、C1Chart.ChartType プロパティを **Candle** に設定して、XYDataSeries.XValues、HighLowSeries.HighValues、HighLowSeries.LowValues、HighLowOpenCloseSeries.OpenValues、および HighLowOpenCloseSeries.CloseValues のデータ値を以下のように指定した場合のローソク足チャートを示しています。

XAML

```
<clchart:C1Chart ChartType="Candle">
  <clchart:C1Chart.Data>
    <clchart:ChartData>
      <clchart:HighLowOpenCloseSeries
        XValues="1 2 3 4 5"
        HighValues="103 105 107 102 99"
        LowValues="100 99 101 98 97"
        OpenValues="100 100 105 100 99"
        CloseValues="102 103 103 99 98"
      />
    </clchart:ChartData>
  </clchart:C1Chart.Data>
</clchart:C1Chart>
```

上の XAML マークアップは、次の図のようなグラフになります。



ローソクの幅を変更する

ローソクの幅を変更するには、次のように `DataSeries.SymbolSize` プロパティを使用します。

XAML

```
ds.SymbolSize = new Size(5, 5);
```

HighLowOpenClose チャート

一般的なチャートタイプと株価チャートの違いは、**HighLowOpenClose** チャートが特殊な型のデータ系列オブジェクト (**HighLowOpenCloseSeries**) を必要とすることです。この型のデータ系列では、各ポイントが1つの期間(通常は1日)に対応し、次の5つの値を含んでいます。

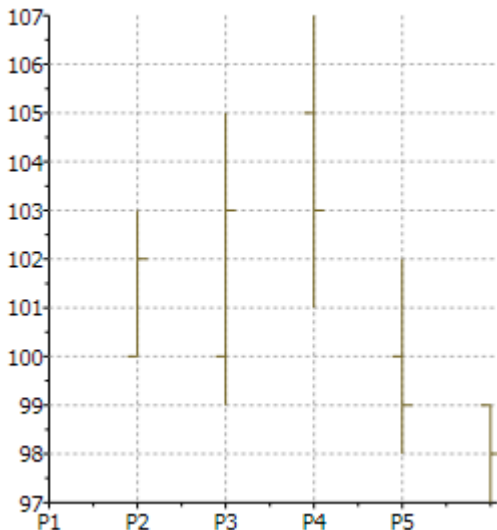
- Time
- 期間の開始時の株価(始値)
- 期間の終了時の株価(終値)
- 期間中の最低株価(安値)
- 期間中の最高株価(高値)

株価チャートを作成するには、これらすべての値を提供する必要があります。

次の図は、`C1Chart.ChartType` プロパティを **HighLowOpenClose** に設定して、`XYDataSeries.XValues`、`HighLowSeries.HighValues`、`HighLowSeries.LowValues`、`HighLowOpenCloseSeries.OpenValues`、および `HighLowOpenCloseSeries.CloseValues` のデータ値を以下のように指定した場合の **HighLowOpenClose** チャートを示しています。

XAML

```
<clchart:C1Chart ChartType="HighLowOpenClose">
  <clchart:C1Chart.Data>
    <clchart:ChartData>
      <clchart:HighLowOpenCloseSeries
        XValues="1 2 3 4 5"
        HighValues="103 105 107 102 99"
        LowValues="100 99 101 98 97"
        OpenValues="100 100 105 100 99"
        CloseValues="102 103 103 99 98"
      />
    </clchart:ChartData>
  </clchart:C1Chart.Data>
</clchart:C1Chart>
```



コードでの Hi-Low-Open-Close チャートの作成

HiLowOpenClose チャートをプログラムで作成するには、次のコードを使用します。

```
C#  
  
HighLowOpenCloseSeries ds = new HighLowOpenCloseSeries()  
{  
    XValueBinding = new System.Windows.Data.Binding("NumberOfDay"),  
    HighValueBinding = new System.Windows.Data.Binding("High"),  
    LowValueBinding = new System.Windows.Data.Binding("Low"),  
    OpenValueBinding = new System.Windows.Data.Binding("Open"),  
    CloseValueBinding = new System.Windows.Data.Binding("Close"),  
    SymbolStrokeThickness = 1,    SymbolSize = new Size(5, 5)  
}  
ds.PlotElementLoaded += (s, e) =>  
{  
    PlotElement pe = (PlotElement)s;  
    double open = (double)pe.DataPoint["OpenValues"];  
    double close = (double)pe.DataPoint["CloseValues"];  
    if (open > close)  
    {  
        pe.Fill = green;  
        pe.Stroke = green;  
    }  
    else  
    {  
        pe.Fill = red;  
        pe.Stroke = red;  
    }  
};
```

たとえば、これらの値がアプリケーションからコレクションとして提供された場合は、以下のコードを使用してデータ系列を作成できます。

C#

```
//データ系列を作成します
HighLowOpenCloseSeries ds = new HighLowOpenCloseSeries();
ds.XValuesSource = dates; //日付は x 軸に置きます
ds.OpenValuesSource = open;
ds.CloseValuesSource = close;
ds.HighValuesSource = hi;
ds.LowValuesSource = lo;
//系列をチャートに追加します
chart.Data.Children.Add(ds);
//グラフタイプを設定します
chart.ChartType = isCandle
    ? ChartType.Candle
    : ChartType.HighLowOpenClose;
```

別の方法として、データ連結を使用できます。たとえば、データが StockQuote オブジェクトのコレクションとしてある場合は、次のようになります。

C#

```
public class Quote
{
    public DateTime Date { get; set; }
    public double Open { get; set; }
    public double Close { get; set; }
    public double High { get; set; }
    public double Low { get; set; }
}
```

次に、データ系列を作成するコードは、次のようになります。

C#

```
//データ系列を作成します
HighLowOpenCloseSeries ds = new HighLowOpenCloseSeries();
//5つの値をすべて連結します
ds.XValueBinding = new Binding("Date"); //日付は x 軸に置きます
ds.OpenValueBinding = new Binding("Open");
ds.CloseValueBinding = new Binding("Close");
ds.HighValueBinding = new Binding("High");
ds.LowValueBinding = new Binding("Low");
//系列をチャートに追加します
chart.Data.Children.Add(ds);
//グラフタイプを設定します
chart.ChartType = isCandle
    ? ChartType.Candle
    : ChartType.HighLowOpenClose;
```

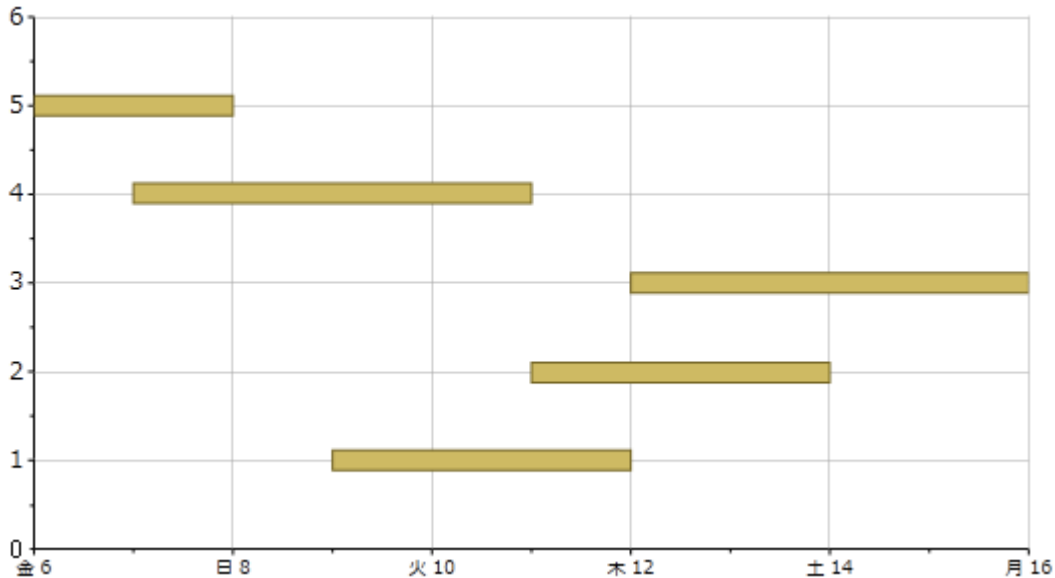
ガントチャート

ガントグラフでは、**HighLowSeries** 型のデータ系列オブジェクトを使用します。各データ系列は1つのタスクを表し、各タスクには一連の開始値と終了値があります。単純なタスクの場合、開始値と終了値はそれぞれ1つです。タスクが複数の逐次サブタ

Chart for WPF/Silverlight

スクで構成されている場合は、開始値と終了値のペアが複数になります。

次の図は、**ガントグラフ**を表します。



ガントグラフを表示するには、まず **Task** オブジェクトを定義します。

C#

```
class Task
{
    public string Name { get; set; }
    public DateTime Start { get; set; }
    public DateTime End { get; set; }
    public bool IsGroup { get; set; }
    public Task(string name, DateTime start, DateTime end, bool isGroup)
    {
        Name = name;
        Start = start;
        End = end;
        IsGroup = isGroup;
    }
}
```

次に、ガントグラフとして表示される一連の Task オブジェクトを作成するメソッドを定義します。

C#

```
Task[] GetTasks()
{
    return new Task[]
    {
        new Task("Alpha", new DateTime(2008,1,1), new DateTime(2008,2,15), true),
        new Task("Spec", new DateTime(2008,1,1), new DateTime(2008,1,15), false),
        new Task("Prototype", new DateTime(2008,1,15), new DateTime(2008,1,31), false),
        new Task("Document", new DateTime(2008,2,1), new DateTime(2008,2,10), false),
        new Task("Test", new DateTime(2008,2,1), new DateTime(2008,2,12), false),
        new Task("Setup", new DateTime(2008,2,12), new DateTime(2008,2,15), false),
    }
}
```

```

new Task("Beta", new DateTime(2008,2,15), new DateTime(2008,3,15), true),
new Task("WebPage", new DateTime(2008,2,15), new DateTime(2008,2,28), false),
new Task("Save bugs", new DateTime(2008,2,28), new DateTime(2008,3,10), false),
new Task("Fix bugs", new DateTime(2008,3,1), new DateTime(2008,3,15), false),
new Task("Ship", new DateTime(2008,3,14), new DateTime(2008,3,15), false),
};
}

```

これでタスクの作成が完了し、**ガントグラフ**を作成する準備ができました。

```

C#
private void CreateGanttChart()
{
    // 現在のグラフをクリア
    clChart.Reset(true);

    // グラフタイプを設定
    clChart.ChartType = ChartType.Gantt;

    // グラフに入力
    var tasks = GetTasks();
    foreach (var task in tasks)
    {
        // タスクあたり1つの系列を作成
        var ds = new HighLowSeries();
        ds.Label = task.Name;
        ds.LowValuesSource = new DateTime[] { task.Start };
        ds.HighValuesSource = new DateTime[] { task.End };
        ds.SymbolSize = new Size(0, task.IsGroup ? 30 : 10);

        // 系列をグラフに追加
        clChart.Data.Children.Add(ds);
    }

    // タスク名を Y 軸に表示
    clChart.Data.ItemNames =
        (from task in tasks select task.Name).ToArray();

    // Y 軸をカスタマイズ
    var ax = clChart.View.AxisY;
    ax.Reversed = true;
    ax.MajorGridStroke = null;

    // X 軸をカスタマイズ
    ax = clChart.View.AxisX;
    ax.MajorGridStrokeDashes = null;
    ax.MajorGridFill = new SolidColorBrush(Color.FromArgb(20, 120, 120, 120));
    ax.Min = new DateTime(2008, 1, 1).ToOADate();
}

```

C1Chart をクリアしてグラフタイプを設定したあと、コードではタスクを列挙して、それぞれに1つの **HighLowSeries** を作成しています。系列の **Label**、**LowValuesSource**、**HighValuesSource** の各プロパティの設定に加えて、コードで

Chart for WPF/Silverlight

は、**SymbolSize** プロパティを使用して各バーの高さを設定します。このサンプルでは、いくつかのタスクをグループタスクとして定義して、それらの高さを通常のタスクより高くしています。

次に、Linq ステートメントを使用してタスク名を抽出し、それらを**ItemNames** プロパティに割り当てます。これによって、**C1Chart** はタスク名を Y 軸に表示するようになります。

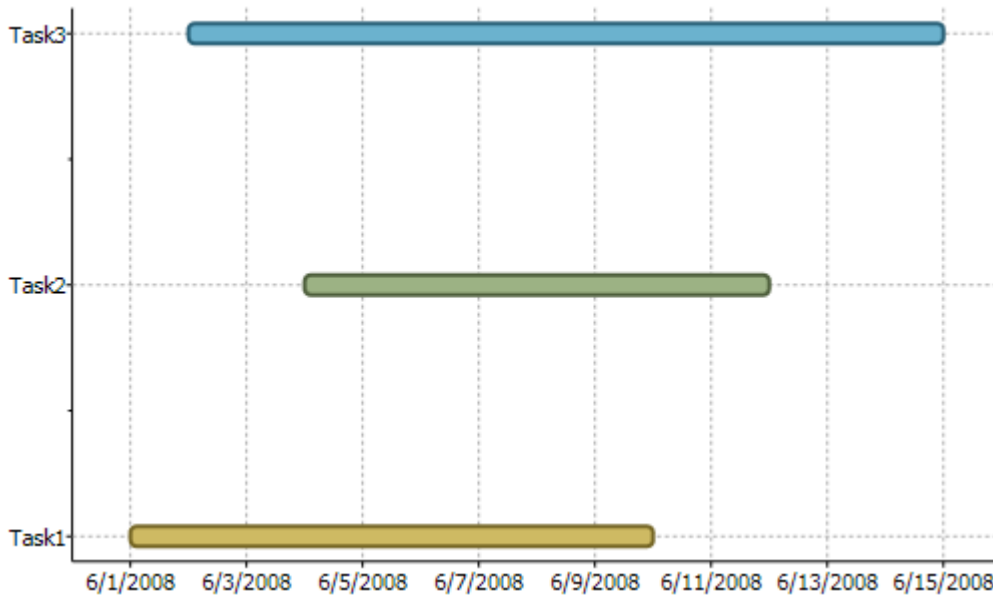
最後に、コードでは軸をカスタマイズしています。Y 軸を反転させて、最初のタスクがグラフの一番上に表示されるようにします。縦のグリッド線と1目盛置きの横線が表示されるように軸を設定しています。

マークアップでガントチャートの作成

ガントチャートを作成するには、次の XAML マークアップを使用します。

XAML

```
<clchart:C1Chart Margin="0" Name="c1Chart1"
    xmlns:sys="clr-namespace:System;assembly=mscorlib">
  <clchart:C1Chart.Resources>
    <x:Array x:Key="start" Type="sys:DateTime" >
      <sys:DateTime>2008-6-1</sys:DateTime>
      <sys:DateTime>2008-6-4</sys:DateTime>
      <sys:DateTime>2008-6-2</sys:DateTime>
    </x:Array>
    <x:Array x:Key="end" Type="sys:DateTime">
      <sys:DateTime>2008-6-10</sys:DateTime>
      <sys:DateTime>2008-6-12</sys:DateTime>
      <sys:DateTime>2008-6-15</sys:DateTime>
    </x:Array>
  </clchart:C1Chart.Resources>
  <clchart:C1Chart.Data>
    <clchart:ChartData>
      <clchart:ChartData.Renderer>
        <clchart:Renderer2D Inverted="True" ColorScheme="Point"/>
      </clchart:ChartData.Renderer>
      <clchart:ChartData.ItemNames>Task1 Task2 Task3</clchart:ChartData.ItemNames>
      <clchart:HighLowSeries HighValuesSource="{StaticResource end}"
        LowValuesSource="{StaticResource start}"/>
    </clchart:ChartData>
  </clchart:C1Chart.Data>
  <clchart:C1Chart.View>
    <clchart:ChartView>
      <clchart:ChartView.AxisX>
        <clchart:Axis IsTime="True" AnnoFormat="d"/>
      </clchart:ChartView.AxisX>
    </clchart:ChartView>
  </clchart:C1Chart.View>
</clchart:C1Chart>
```

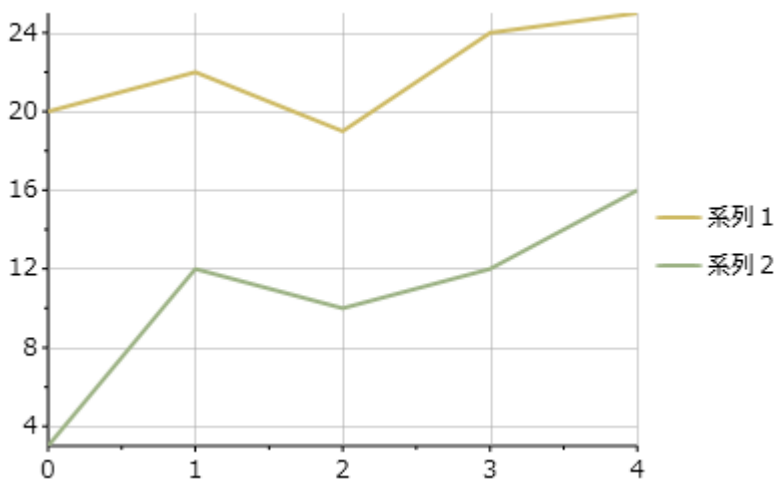
折れ線グラフ

Chart for WPF/Silverlight では、以下のタイプの折れ線グラフがサポートされています。

- Line
- LineSmoothed
- LineStacked
- LineStacked100pc
- LineSymbols
- LineSymbolsSmoothed
- LineSymbolsStacked
- LineSymbolsStacked100pc

折れ線グラフ

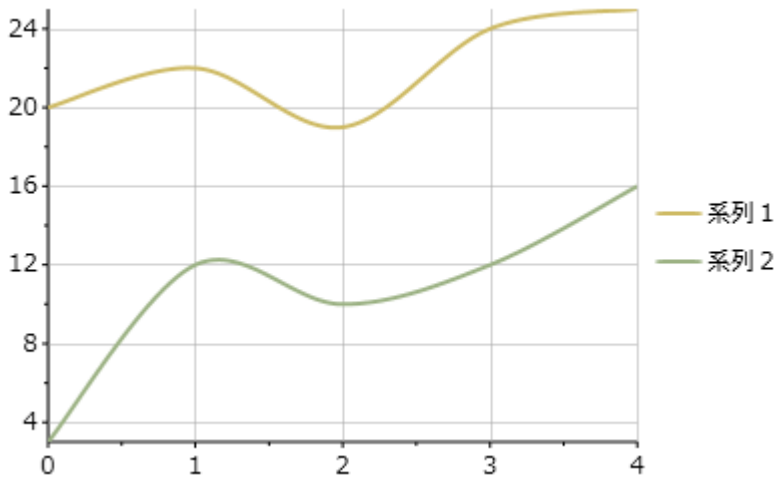
次の図は、ChartType プロパティを Line に設定したときの折れ線グラフを表します。



平滑化折れ線グラフ

次の図は、ChartType プロパティを LineSmoothed に設定したときの平滑化折れ線グラフを表します。

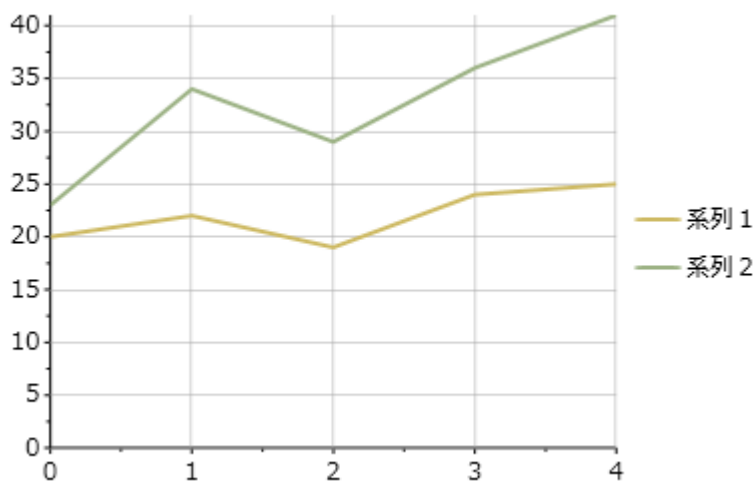
Chart for WPF/Silverlight



積み重ね折れ線グラフ

特定の積み重ね折れ線グラフを作成するには、**ChartType** 列挙から **LineStacked** メンバを選択します。積み重ねグラフは、各系列の値を前の系列の値の上に積み重ねることによってデータを表します。

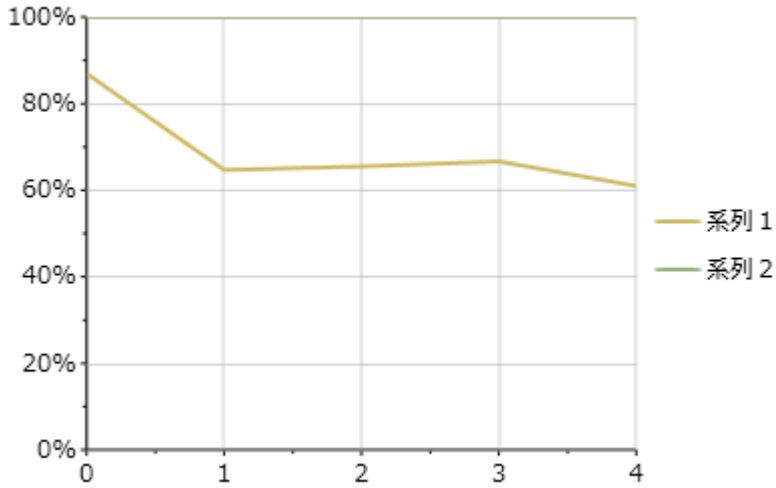
次の図は、**ChartType** プロパティを **LineStacked** に設定したときの積み重ね折れ線グラフを表します。



100% 積み重ね折れ線グラフ

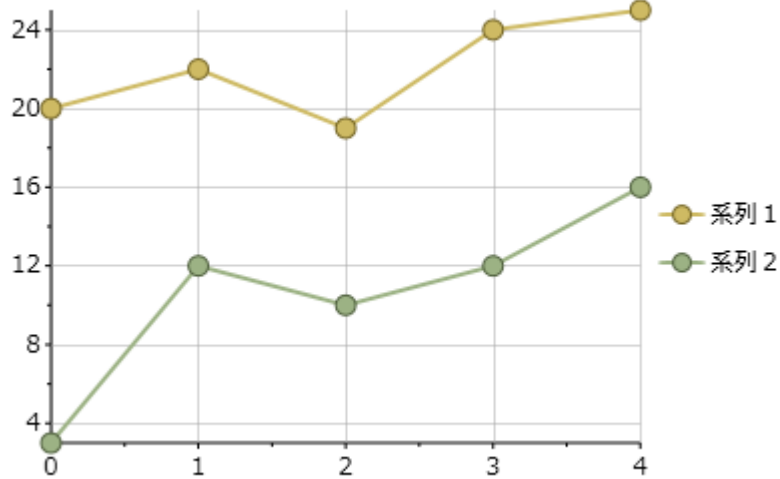
特定の積み重ね折れ線グラフを作成するには、**ChartType** 列挙から **LineStacked100pc** メンバを選択します。積み重ねグラフは、各系列の値を前の系列の値の上に積み重ねることによってデータを表します。

次の図は、**ChartType** プロパティを **LineStacked100pc** に設定したときの 100% 積み重ね折れ線グラフを表します。



記号付き折れ線グラフ

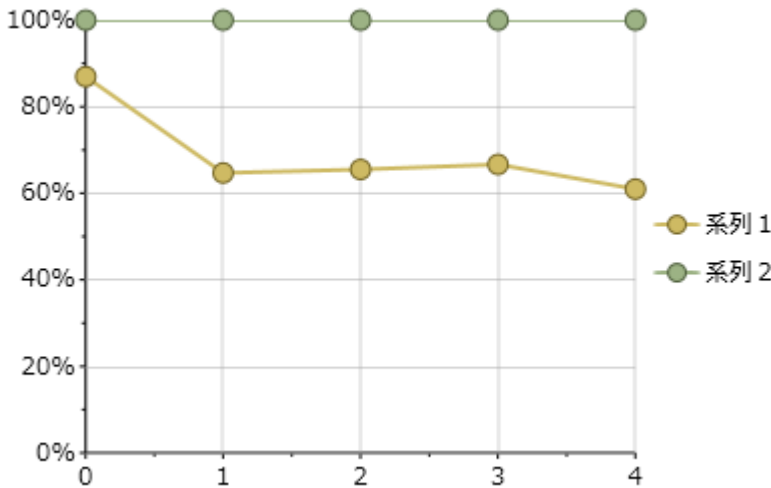
次の図は、**ChartType** プロパティを **LineSymbols** に設定したときの **記号付き折れ線** グラフを表します。



100% 記号付き積み重ね折れ線グラフ

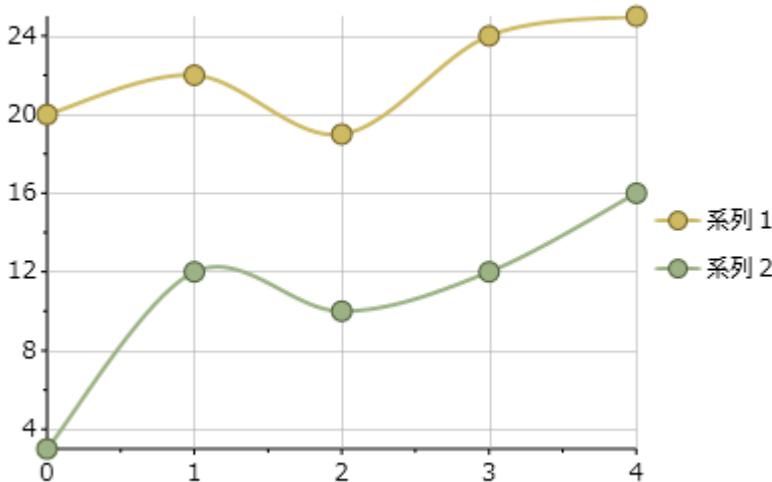
次の図は、**ChartType** プロパティを **LineSymbolsStacked100pc** に設定したときの **記号付き積み重ね折れ線** グラフを表します。

Chart for WPF/Silverlight



記号付き平滑化折れ線グラフ

次の図は、**ChartType** プロパティを **LineSymbolsSmoothed** に設定したときの記号付き平滑化折れ線グラフを表します。



円グラフ

円グラフは、単純な値を表示するために広く使用されています。円グラフは視覚的なアピール力に優れており、陰影や回転といった 3D 効果を伴って表示されることがよくあります。

円グラフには、**C1Chart**の他のグラフタイプに比べて際立った違いがあります。円グラフでは、各系列がその円の1スライスを表します。そのため、系列が1つしかない円グラフはあり得ません(それは単なる円になります)。ほとんどの場合、円グラフには(スライスあたり1系列で)複数の系列が含まれ、各系列にはデータポイントが1つしかありません。**C1Chart**では、複数のデータポイントを持つ系列はグラフ内で複数の円として表されます。

XAML マークアップを使用して円グラフを作成する場合、マークアップは次のようになります。

XAML

```
<c1chart:C1Chart Name="c1Chart1" ChartType="Pie">
  <c1chart:C1Chart.Data>
    <c1chart:ChartData>
      <c1chart:ChartData.ItemNames>P1 P2 P3 P4 P5</c1chart:ChartData.ItemNames>
    </c1chart:ChartData>
  </c1chart:C1Chart.Data>
</c1chart:C1Chart>
```

```

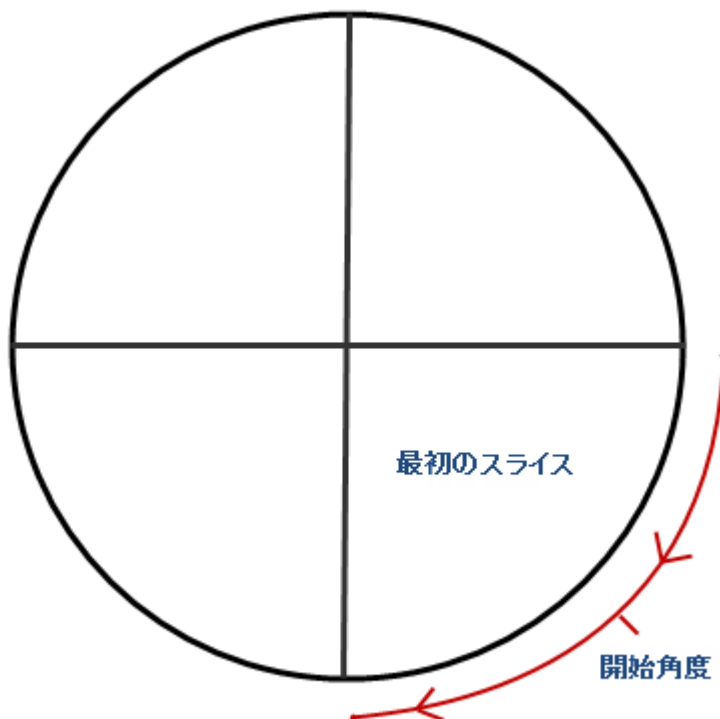
        <clchart:DataSeries Values="20 22 19 24 25" />
    </clchart:ChartData>
</clchart:C1Chart.Data>
    <clchart:C1ChartLegend DockPanel.Dock="Right" />
</clchart:C1Chart>

```

円グラフには、円グラフコントロールのカスタマイズに役立つ、特別なプロパティがあります。SetStartingAngle プロパティまたは Direction プロパティを使用すると、円グラフの最初の開始角度を変更したり、スライスの変更することができます。また、BasePieRenderer プロパティを使用して、メインチャートから1つのスライスを分離して強調表示することもできます。

開始角度

PieOptions.SetStartingAngle プロパティは、円グラフ内の最初のスライスの位置を定義します。最初のスライスは、常に 90 度から始まります。開始角度は、90 度から時計回りに計測します。



PieOptions.SetStartingAngle プロパティを使用して、最初の系列のスライスの開始角度を指定します。次の図は、開始角度を 90 に設定したところを示します。

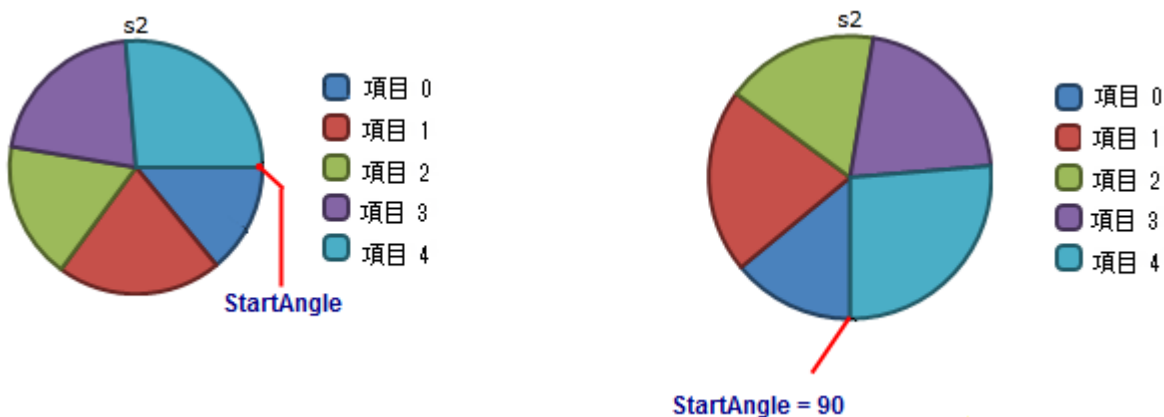


Chart for WPF/Silverlight

円の分割

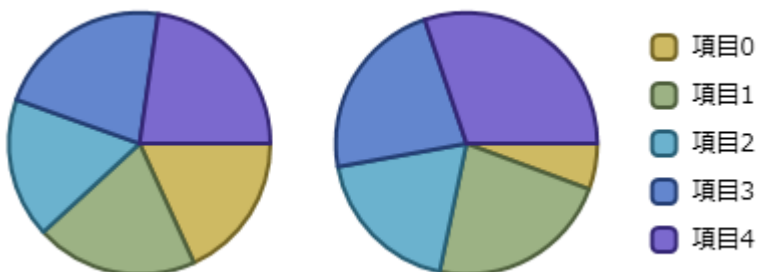
円グラフのスライスは、分割することによって強調できます。分割したスライスは、円の残りの部分から浮き出て表示されます。系列の**Offset**プロパティを使用して、分割したスライスの円の中心からのオフセットを設定します。オフセットは、円の半径に対する割合で測定されます。

Chart for WPF/Silverlight では、以下のタイプの円グラフがサポートされています。

- Pie
- PieStacked
- 3D Pie (WPF のみ)
- 3D Doughnut Pie (WPF のみ)
- 3D Exploded Pie (WPF のみ)
- 3D Exploded Doughnut Pie (WPF のみ)
- Doughnut Pie
- Exploded Pie
- Exploded Doughnut Pie

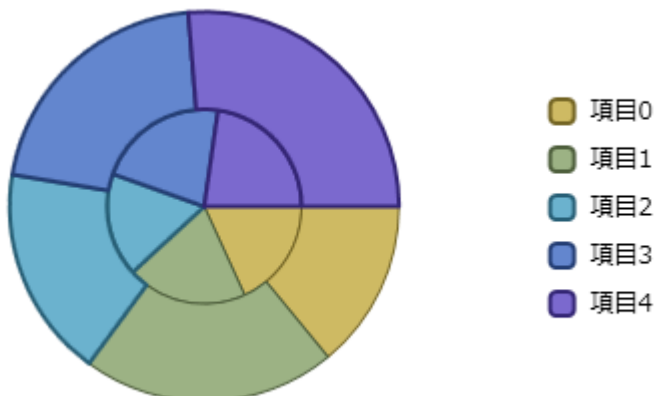
円グラフ

次の図は、**ChartType**プロパティを**Pie**に設定したときの円グラフを表します。



積層円グラフ

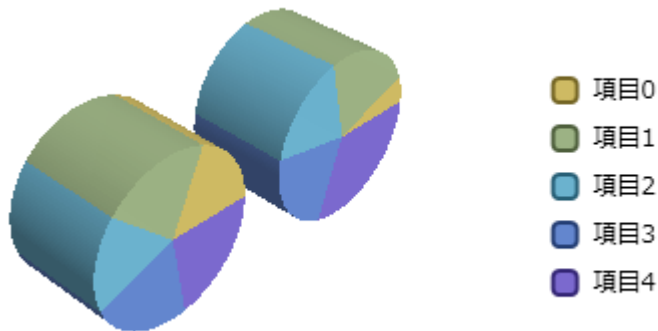
次の画像は、**ChartType**プロパティを**PieStacked**に設定した場合の積層円グラフを示しています。



3D 円グラフ (WPF のみ)

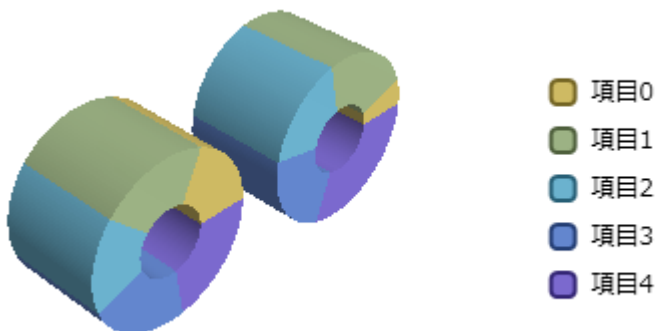
Pie3Dクラスは、3D 円グラフのプロット要素に関連付けられたデータポイントにアクセスする、マウスポインタがプロット要素上に置かれているときにその要素の値を取得する、プロット要素のサイズ(ピクセル)を取得または設定する、各ポイントを滑らかな線で接続するかどうかを指定するといった場合に使用します。

次の図は、ChartTypeプロパティをPie3Dに設定したときの3D 円グラフを表します。



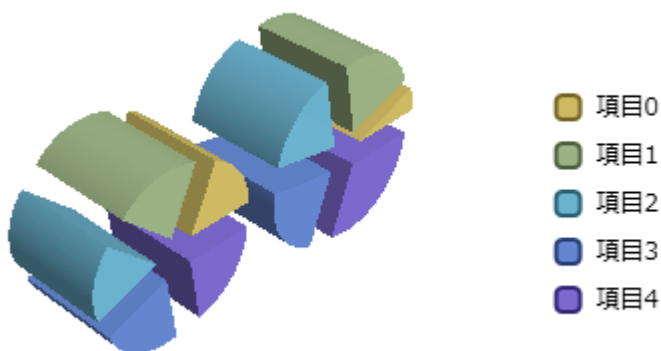
3D ドーナツ円グラフ (WPF のみ)

次の図は、ChartTypeプロパティを Pie3DDoughnut に設定したときの3D ドーナツ円グラフを表します。



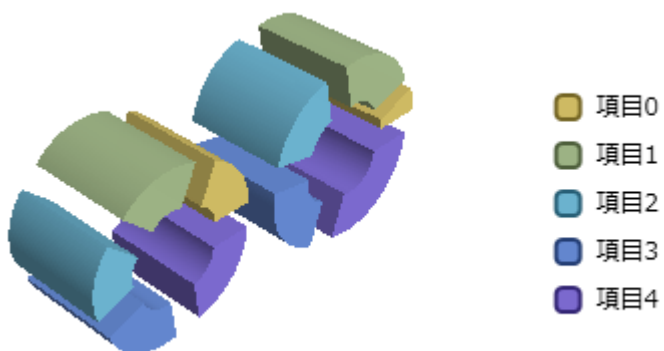
3D 分割円グラフ (WPF のみ)

次の図は、ChartTypeプロパティを Pie3DExploded に設定したときの3D 分割円グラフを表します。



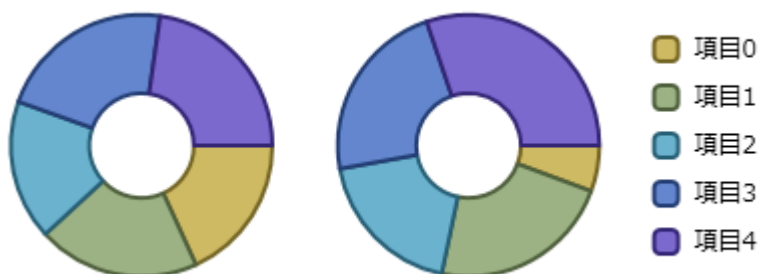
3D 分割ドーナツ円グラフ (WPF のみ)

次の図は、ChartTypeプロパティを Pie3DExplodedDoughnut に設定したときの3D 分割ドーナツ円グラフを表します。



ドーナツ円グラフ

次の図は、`ChartType`プロパティを **PieDoughnut** に設定したときのドーナツ円グラフを表します。



分割円グラフ

次の図は、`ChartType`プロパティを **PieExploded** に設定したときの分割円グラフを表します。



分割ドーナツ円グラフ

次の図は、`ChartType`プロパティを **PieExplodedDoughnut** に設定したときの分割ドーナツ円グラフを表します。



円グラフに重ならないように接続線を追加する

次の XAML コードのように、PlotElement.LabelLine 添付プロパティを使用して接続線を追加できます。

XAML

```
<cl:DataSeries.PointLabelTemplate>
  <DataTemplate>
    <Border BorderBrush="DarkGray" BorderThickness="1" Background="LightGray"
      cl:PlotElement.LabelAlignment="Auto"
      cl:PlotElement.LabelOffset="30,0">
      <TextBlock Text="{Binding Value, StringFormat=0}" />
      <cl:PlotElement.LabelLine>
        <Line Stroke="LightGray" StrokeThickness="2" />
      </cl:PlotElement.LabelLine>
    </Border>
  </DataTemplate>
</cl:DataSeries.PointLabelTemplate>
```

円グラフへのラベルの追加

円グラフのラベルに複数の値を追加するには、次のようなラベルテンプレートを作成します。

XAML

```
<clchart:C1Chart Name="c1Chart1" ChartType="Pie">
  <clchart:C1Chart.Resources>
    <DataTemplate x:Key="lbl">
      <StackPanel>
        <StackPanel Orientation="Horizontal">
          <TextBlock Text="{Binding Path=Name}" />
          <TextBlock Text="=" />
          <TextBlock Text="{Binding Path=Value}" />
        </StackPanel>
        <TextBlock Text="{Binding Path=PercentageSeries, Converter={x:Static
clchart:Converters.Format}, ConverterParameter=#.##}" />
      </StackPanel>
    </DataTemplate>
  </clchart:C1Chart.Resources>
  <clchart:C1Chart.Data>
```

Chart for WPF/Silverlight

```
<clchart:ChartData>
  <clchart:ChartData.ItemNames>P1 P2 P3 P4 P5</clchart:ChartData.ItemNames >
  <clchart:DataSeries Values="20 22 19 24 25" PointLabelTemplate="{StaticResource lbl}" />
</clchart:ChartData>
</clchart:C1Chart.Data>
<clchart:C1ChartLegend DockPanel.Dock="Right" />
</clchart:C1Chart>
```

すべてのセグメントのオフセットを変更する

円グラフのすべてのセグメントのオフセットを変更するには、次のコードを使用します。

```
chart.DataContext = new double[] { 1, 2, 3 };
chart.ChartType = ChartType.Pie;
chart.Loaded += (s, e) => ((BasePieRenderer)chart.Data.Renderer).Offset = 0.1;
```


特定のセグメントのオフセットを変更することもできますが、それには、**PlotElementLoaded** イベントで手作業で位置を変更する必要があります。

3D 円グラフのデフォルトの表示角度を設定する (WPF のみ)

3D 円グラフのデフォルトの表示角度を設定するには、次のコードを使用します。

XAML

```
chart.View.Camera.Transform = new RotateTransform3D(new AxisAngleRotation3D(new Vector3D(0,0,1),45));
```

 **メモ:**このトピックの内容は、ComponentOne Studio for WPF にのみ適用されます。

ポーラチャート

ポーラチャートは、系列ごとに X 座標と Y 座標を (theta,r) として描画します。

- theta 値 - チャートの原点からの回転角度。theta は、度(デフォルト)またはラジアン単位で指定できます。
- r - チャートの原点からの距離。

X 軸は円なので、X 軸の最大および最小値は固定されています。

ポーラチャートを同じチャート領域にある他のグラフタイプと結合することはできません。

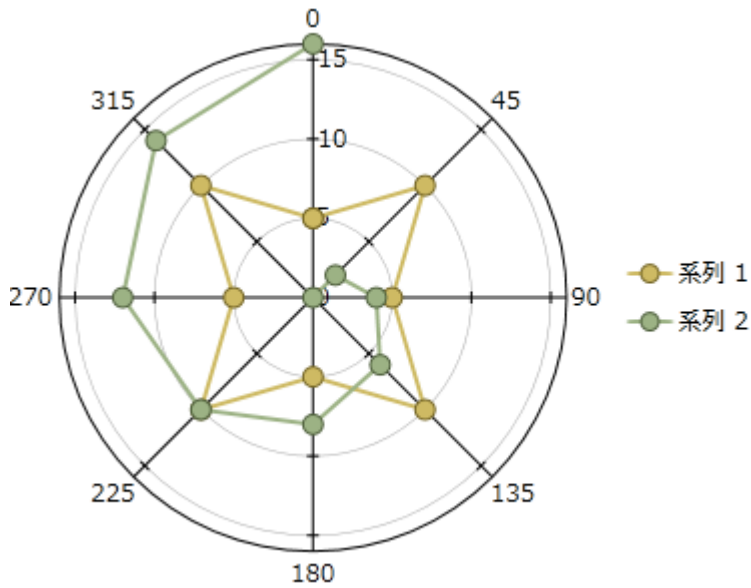
以下の XAML マークアップは XYDataSeries のデータ値を指定し、マークアップの直後にある図の作成に使用されます。

XAML

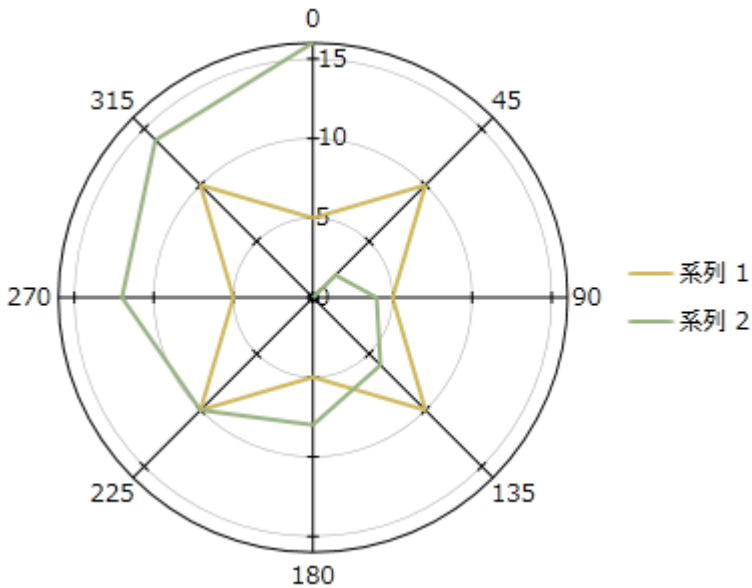
```
<clchart:C1Chart Name="c1Chart1" ChartType="PolarLinesSymbols">
  <clchart:C1Chart.Data>
    <clchart:ChartData>
      <clchart:XYDataSeries Label="系列1" Values="5 10 5 10 5 10 5 10 5"
        XValues="0 45 90 135 180 225 270 315 0"/>
      <clchart:XYDataSeries Label="系列2" Values="0 2 4 6 8 10 12 14 16"
        XValues="0 45 90 135 180 225 270 315 0"/>
    </clchart:ChartData>
  </clchart:C1Chart.Data>
</clchart:C1Chart>
```

```

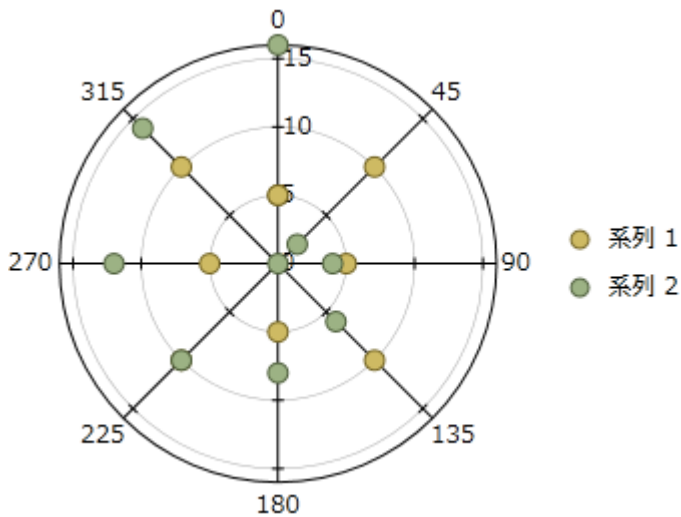
</clchart:ChartData>
</clchart:C1Chart.Data>
<clchart:C1ChartLegend DockPanel.Dock="Right" /> </clchart:C1Chart>
    
```



次の画像は、**ChartType** プロパティを **PolarLines** に設定した場合のポーラチャート(シンボルおよびライン付き)を示しています。

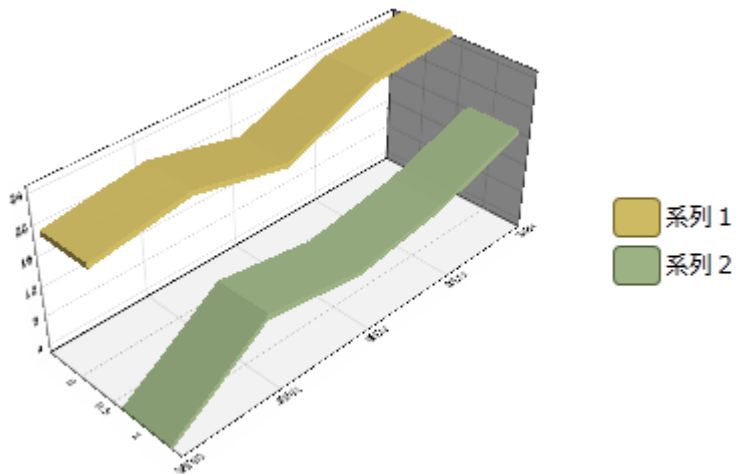



次の画像は、**ChartType** プロパティを **PolarSymbols** に設定した場合のポーラチャートを示しています。



3D リボングラフ (WPF のみ)

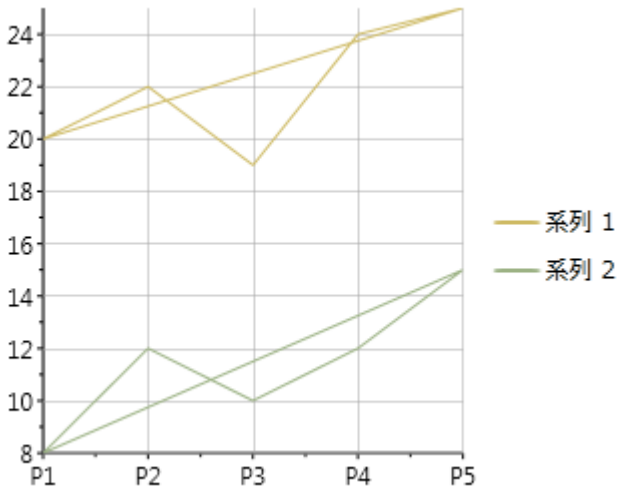
次の図は、**ChartType** プロパティを **Ribbon** に設定したときの **3D リボングラフ**を表します。



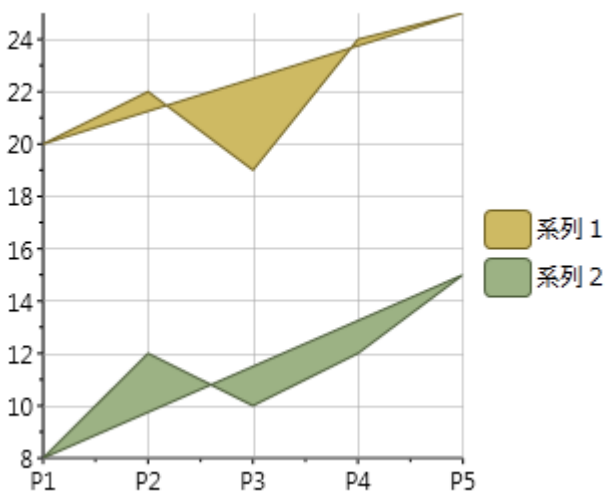
 **メモ:**このトピックの内容は、ComponentOne Studio for WPF にのみ適用されます。

多角形グラフ

次の画像は、**ChartType** プロパティを **Polygon** に設定した場合の **多角形グラフ**を示しています。



次の画像は、**ChartType** プロパティを **PolygonFilled** に設定した場合の**多角形塗りつぶし**グラフを示しています。



レーダーチャート

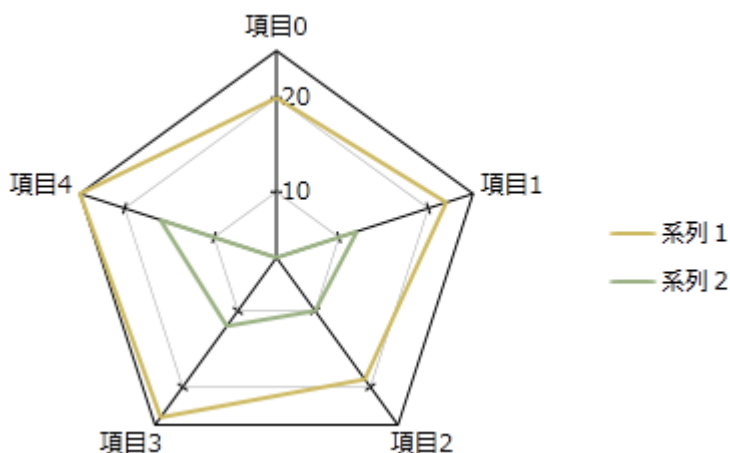
レーダーチャートは、ポーラチャートのバリエーションです。レーダーチャートは、データセットごとにレーダー線に沿って y 値を描画します。データに一意のポイントが n 個ある場合、チャート面は、n 個の等しい角度のセグメントに分割され、 $360/n$ 度の間隔でレーダー線 (各ポイントを表す) が描画されます。デフォルトでは、最初のポイントを表すレーダー線が垂直に描画されます (90 度)。

レーダーチャートのラベルは、**ItemNames** プロパティを使って設定できます。これらのラベルは、各放射線の末端に配置されます。

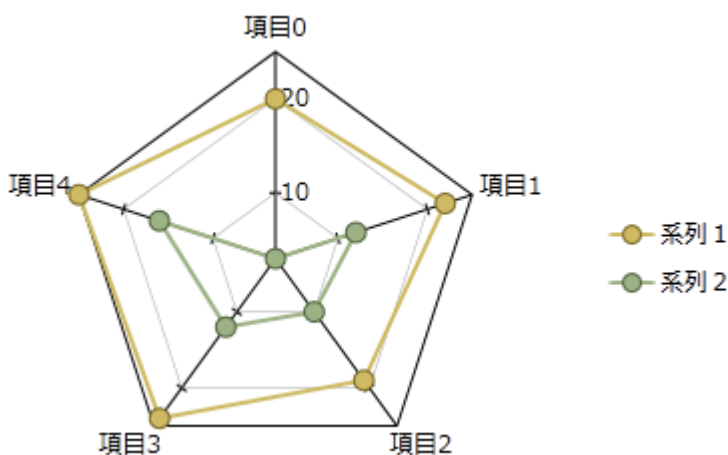
開始角度の設定

PolarRadarOptions クラスの **PolarRadarOptions.SetStartingAngle** 添付プロパティは、レーダーチャートの開始角度を設定します。開始角度を使用して、チャートを時計回りに回転できます。たとえば、**SetStartingAngle** プロパティを 90 に設定すると、チャートが時計回りに 90 度回転します。

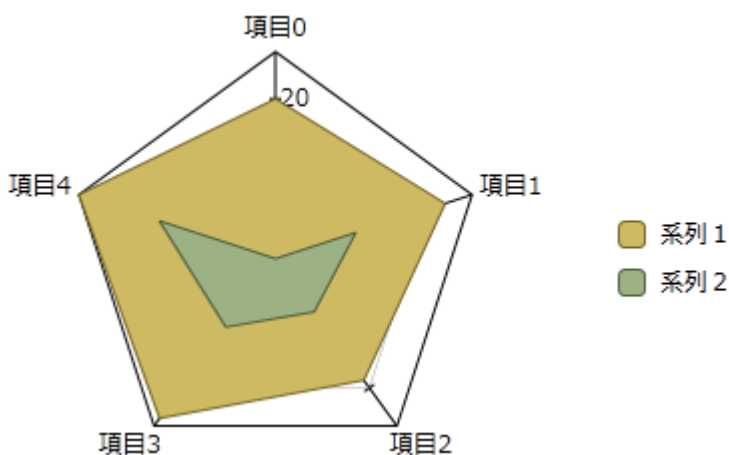
次の図は、**ChartType** プロパティを **Radar** に設定した場合の**レーダーチャート**を示しています。



次の画像は、**ChartType** プロパティを **RadarSymbols** に設定した場合のレーダーチャート(シンボル付き)を示しています。



次の画像は、**ChartType** プロパティを **RadarFilled** に設定した場合の塗りつぶしレーダーチャートを示しています。



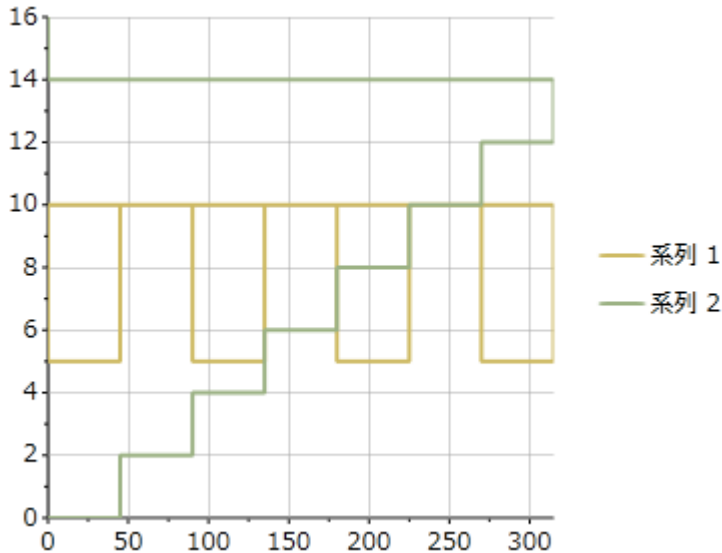
階段グラフ

階段グラフは、**XYPlot** グラフの形式の1つです。**階段グラフ**は、**X** の値が特定の値になると **Y** の値が突然変化するという、値が離散的に変化する場合によく使用されます。最もわかりやすい例としては、時間の経過に伴う小切手口座の残高の変化があります。時間の経過 (**X** 値) と共に、預金が発生したり、小切手が切られたりします。そのたびに、小切手口座の残高 (**Y** 値) は、徐々にではなく突発的に変更されます。時間が経過しても預金が発生せず小切手も切られない場合、その期間の残高 (**Y** 値) は変わりません。

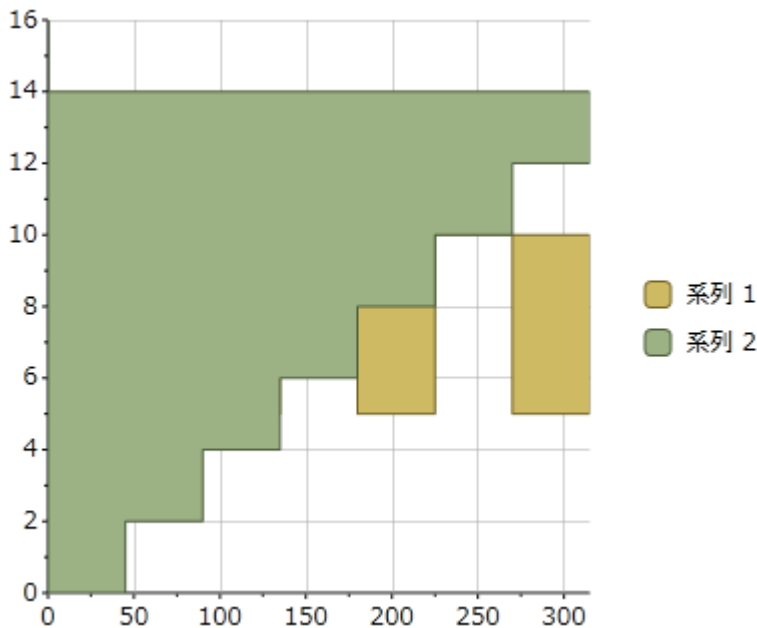
折れ線グラフまたはXYPlot グラフと同様に、階段グラフの外観は、各系列の**Connection** プロパティおよび**Symbol** プロパティを使って色、シンボルサイズ、線の太さを変更することでカスタマイズできます。シンボルを完全に削除してポイント間の関係を強調することも、シンボルを含めて実際のデータ値を示すこともできます。階段グラフでは、データ欠損がある場合でも、データ欠損の X 値までは既知の情報を示す系列線によって問題なく作成されます。シンボルと線は、データ欠損が終了した時点から再度描画されます。

ほとんどの XY スタイルのプロットと同様に、**階段グラフ**は必要に応じて積み上げることができます。

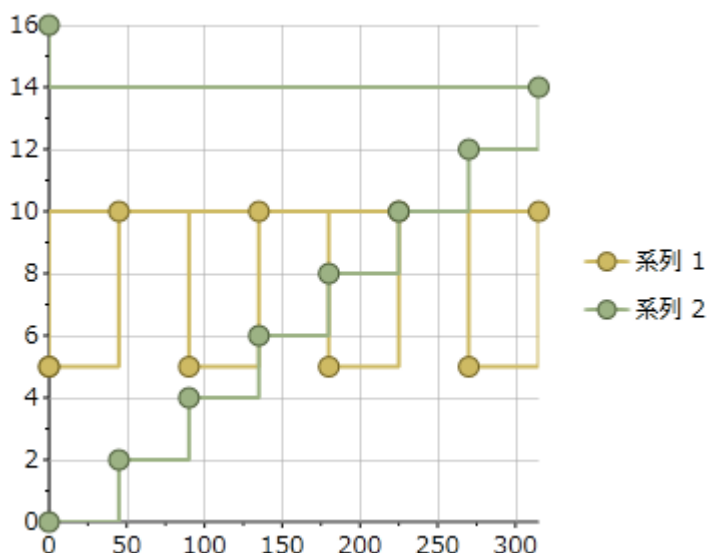
次の画像は、**ChartType** プロパティを **Step** に設定した場合の階段グラフを示しています。



次の画像は、**ChartType** プロパティを **StepArea** に設定した場合の**階段エリアグラフ**(シンボル付き)を示しています。



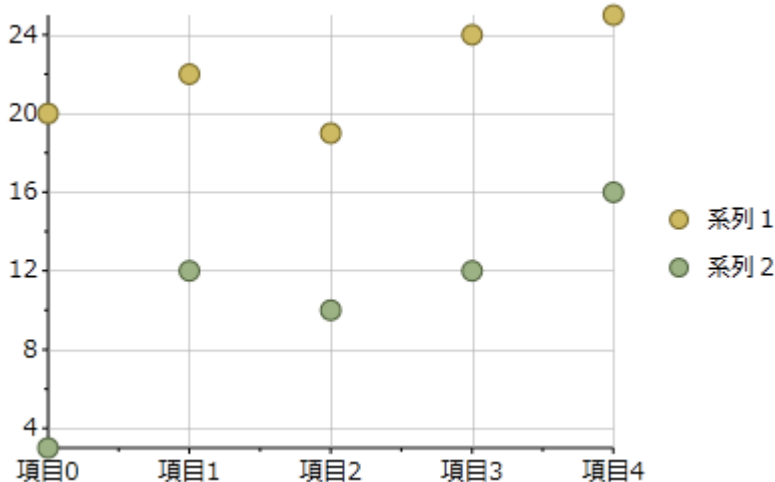
次の画像は、**ChartType** プロパティを **StepSymbols** に設定した場合の**階段グラフ**(シンボル付き)を示しています。



散布グラフ

XY プロットグラフは、**散布図**とも呼ばれています。

次の図は、**ChartType** プロパティを **XYPlot** に設定したときの **XY プロットグラフ**を表します。



散布グラフの作成

XY グラフ(散布図)は、変数間の関係を示すために使用されます。これまでに紹介したグラフと異なり、XY グラフでは、各ポイントに2つの数値があります。それらの値の1つを X 軸に、もう1つを Y 軸にプロットすることで、グラフは1つの変数に対するもう1つの変数の影響を示します。

前に作成した同じデータを使用して **C1Chart** の説明を続行しますが、今回は、2つの製品の収益の間の関係を示すXY グラフを作成します。たとえば、**ウィジェット**の高収益が**ガジェット**の高収益に関係しているかどうか(製品が連動してうまく機能している可能性)や、**ウィジェット**の高収益が**ガジェット**の低収益に関係しているかどうか(製品の1つを購入しているユーザーが実はその他の製品を必要としていない可能性)を確認することが必要な場合があります。

これを行うには、以前と同じ手順を実行します。主な違いは、今回、より単純な **DataSeries** オブジェクトではなく、**XYDataSeries** オブジェクトをグラフの **Data.Children** コレクションに追加することです。データの取得に使用される Linq ステートメントも、少し洗練され、興味深いものになっています。

手順1: グラフタイプの選択

このコードは、既存の系列をすべてクリアして、グラフタイプを設定します。

```
public Window1()
{
    InitializeComponent();
    // 現在のグラフをクリア
    clChart.Reset(true);
    // グラフタイプを設定
    clChart.ChartType = ChartType.XYPlot;
}
```

手順2:軸の設定

今回は XY 系列を作成するため、値の軸が2つあります(以前は、ラベルの軸と値の軸が1つずつでした)。以前行ったのと同様、タイトルと書式を両方の軸に関連付けます。スケールと注釈の書式も以前のように設定します。また、**AnnoAngle** プロパティを使用して X 軸の各注釈ラベルを回転させて、それらが重ならないようにします。

```
// 軸を取得
var yAxis = _clChart.View.AxisY;
var xAxis = _clChart.View.AxisX;

// Y 軸を設定
yAxis.Title = CreateTextBlock("ウィジェットの高収益", 14, FontWeights.Bold);
yAxis.AnnoFormat = "#,##0 ";
yAxis.AutoMin = false;
yAxis.Min = 0;
yAxis.MajorUnit = 2000;
yAxis.AnnoAngle = 0;

// X 軸を設定
xAxis.Title = CreateTextBlock("ガジェットの高収益", 14, FontWeights.Bold);
xAxis.AnnoFormat = "#,##0 ";
xAxis.AutoMin = false;
xAxis.Min = 0;
xAxis.MajorUnit = 2000;
xAxis.AnnoAngle = -90; // 注釈を回転
```

手順3:1つ以上のデータ系列の追加

今回も、前に定義した2つめのデータ提供メソッドを使用します。

```
// データを取得
var data = GetSalesPerMonthData();
```

次に、各日付の**ウィジェット**と**ガジェット**の収益合計に対応する XY のペアを取得する必要があります。Linq を使用して、この情報をデータから直接取得できます。

```
// 売上日付でデータをグループ化
var dataGrouped = from r in data
    group r by r.Date into g
    select new
    {
```

Chart for WPF/Silverlight

```
Date = g.Key, // 日付でグループ化
Widgets = (from rp in g // ウィジェットの収益を追加
           where rp.Product == "ウィジェット"
           select g.Sum(p => rp.Revenue)).Single(),
Gadgets = (from rp in g // ガジェットの収益を追加
           where rp.Product == "ガジェット"
           select g.Sum(p => rp.Revenue)).Single(),
};

// ウィジェットの売上でデータをソート
var dataSorted = from r in dataGrouped
orderby r.Gadgets
select r;
```

1つめの Linq クエリーでは、まずデータを **Date** でグループ化しています。次に各グループについて、関心のある製品ごとに Date とその日付内の収益の合計を含むレコードを作成します。結果は、**Date**、**Widgets**、**Gadgets** の3つのプロパティを持つオブジェクトのリストです。こうしたデータのグループ化や集計は、Linq の強力な機能の1つです。2つめの Linq クエリーは、**ガジェット**の収益でデータをソートするだけです。これらは X 軸上にプロットされる値であり、昇順にする必要があります。記号のみを表示した場合 (**ChartType = XYPlot**) は、ソートされていない値をプロットしても見栄えは悪くありませんが、**Line** や **Area** といったその他のグラフタイプを選択した場合は、ソートしないと乱雑に見えます。データのグループ化、集計、およびソートが正しく完了したら、必要な作業は、データ系列を1つ作成して、1つの値セットを **ValuesSource** プロパティと **XValuesSource** プロパティに割り当てることだけです。

```
// 新しい XYDataSeries を作成
var ds = new XYDataSeries();
// 系列のラベル(C1ChartLegend に表示)を設定
ds.Label = "収益:\r\nウィジェット 対 ガジェット";
// Y の値を入力
ds.ValuesSource = (
from r in dataSorted
select r.Widgets).ToArray();

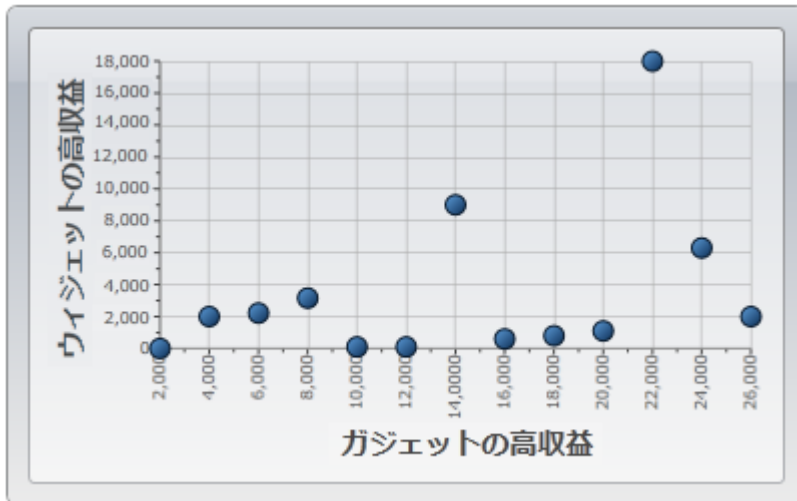
// X の値を入力
ds.XValuesSource = (
from r in dataSorted
select r.Gadgets).ToArray();
// 系列をグラフに追加
c1Chart.Data.Children.Add(ds);
```

手順4:グラフの外観の調整

今回も、最後に Theme プロパティを設定して、グラフの外観を手早く設定します。

```
c1Chart.Theme = c1Chart.TryFindResource(new
ComponentResourceKey(typeof(C1.WPF.C1Chart.C1Chart), "Office2007Black")) as
ResourceDictionary;
}
```

このコードは、プログラムを実行して、**ChartType** プロパティを **XYPlot**、**LineSymbols**、または **Area** に変更し、別種のグラフを作成することによってテストできます。結果は、以下の図のようになるはずですが。

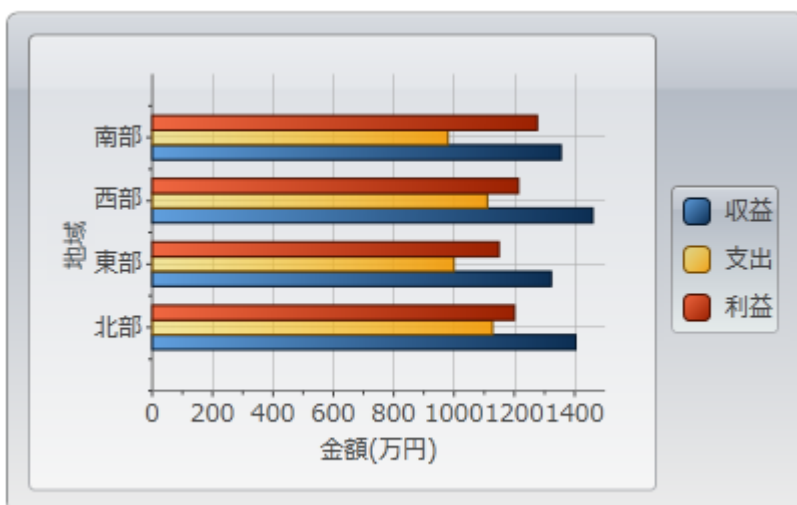


この場合に最適なグラフタイプは、最初の **XYPlot** です。グラフは、**ガジェット**と**ウィジェット**の収益の間にある正の相関を示しています。

これで、基本的なグラフ作成のトピックは終わりです。以上の方法で、一般的なグラフについてはすべてのタイプを作成できます。

単純なグラフ

最も単純なグラフは、各データポイントに数値が1つだけ関連付けられているものです。典型的な例は、次のグラフのような、各地域の販売データを示すグラフです。



グラフを作成するには、まずグラフとして示されるデータを作成する必要があります。次に、必要なデータを作成するコードをいくつか示します。

注意: このコードには、グラフに特有の要素はありません。これは汎用のデータに過ぎません。次のトピックでもこのデータを使用して、時系列グラフおよびXYグラフを作成します。

C#

```
// ダミーの販売データを保持する単純なクラス
public class SalesRecord
{
    // プロパティ
    public string Region { get; set; }
}
```

Chart for WPF/Silverlight

```
public string Product { get; set; }
public DateTime Date { get; set; }
public double Revenue { get; set; }
public double Expense { get; set; }
public double Profit { get { return Revenue - Expense; } }
// コンストラクタ1
public SalesRecord(string region, double revenue, double expense)
{
    Region = region;
    Revenue = revenue;
    Expense = expense;
}
// コンストラクタ2
public SalesRecord(DateTime month, string product, double revenue, double
expense)
{
    Date = month;
    Product = product;
    Revenue = revenue;
    Expense = expense;
}
}
// 地域ごとに1つの SalesRecord を含むリストを返す
List GetSalesPerRegionData()
{
    var data = new List();
    Random rnd = new Random(0);
    foreach (string region in "北部, 東部, 西部, 南部".Split(','))
    {
        data.Add(new SalesRecord(region, 100 + rnd.Next(1500), rnd.Next(500)));
    }
    return data;
}
// 製品ごとに1つの SalesRecord を含むリストを返す、// 期間は 12 カ月
List GetSalesPerMonthData()
{
    var data = new List();
    Random rnd = new Random(0);
    string[] products = new string[] { "ウィジェット", "ガジェット", "スプロケット" };
    for (int i = 0; i < 12; i++)
    {
        foreach (string product in products)
        {
            data.Add(new SalesRecord(
                DateTime.Today.AddMonths(i - 24),
                product,
                rnd.NextDouble() * 1000 * i,
                rnd.NextDouble() * 1000 * i));
        }
    }
    return data;
}
```

```
}
}
```

SalesData クラスがパブリックであることに注意してください。これはデータバインディングのために必要です。グラフの作成では、以下の主な4段階の手順を実行します。

手順1: グラフタイプの選択 このコードは、既存の系列をすべてクリアして、グラフタイプを設定します。

```
C#
public Window1 ()
{
    InitializeComponent ();
    // 現在のグラフをクリア
    clChart.Reset (true);
    // グラフタイプを設定
    clChart.ChartType = ChartType.Bar;
}
```

手順2: 軸の設定

まず、両方の軸への参照を取得します。ほとんどのグラフでは、水平の軸(X)には各ポイントに関連付けられたラベルを表示し、垂直の軸(Y)には値を表示します。例外は、水平の棒を表示する横棒グラフのタイプです。このグラフタイプの場合、ラベルはY軸に表示され、X軸には値が表示されます。

次に、タイトルを軸に割り当てます。軸のタイトルは単なるテキストではなく、**UIElement** オブジェクトです。これは、タイトルの書式を完全に制御できるということです。実は、軸のタイトルには、ボタン、テーブル、または画像を備えた複雑な要素を使用できます。この場合は、後で説明する **CreateTextBlock** メソッドで作成された単純な **TextBlock** 要素を使用します。

また、値の軸を設定してゼロで始まるようにし、桁区切り文字を使用して目盛記号の横に注釈が表示されるようにします。

```
C#
// ラベルの軸を設定
labelAxis.Title = CreateTextBlock ("地域", 14, FontWeights.Bold);

// 値の軸を設定
_clChart.View.AxisX.Title = CreateTextBlock ("金額 (万円)", 14, FontWeights.Bold);
_clChart.View.AxisX.AutoMin = false;
_clChart.View.AxisX.Min = 0;
_clChart.View.AxisX.MajorUnit = 200;
_clChart.View.AxisX.AnnoFormat = "#,##0 ";
```

手順3: つ以上のデータ系列の追加

ここでは、まず前に使用したメソッドを使用してデータを取得します。

```
C#
// データを取得
var data = GetSalesPerRegionData ();
```

次に、ラベルの軸に地域を表示します。これを行うには、各レコードの **Region** プロパティを取得する Linq ステートメントを使用します。結果は配列に変換され、**ItemNames** プロパティに割り当てられます。

```
C#
```

Chart for WPF/Silverlight

```
// ラベルの軸に地域を表示
c1Chart.Data.ItemNames = (from r in data select r.Region).ToArray();
```

Linq の使用によって、コードがいかに直接的かつ簡潔になるかに注意してください。このサンプルデータには地域あたり1つのレコードしか含まれていないため、事情はさらに単純になっています。より現実的なシナリオでは、各地域に複数のレコードがあるため、より複雑な Linq ステートメントを使用して地域ごとにデータをグループ化します。

これで、グラフに追加される実際の **DataSeries** オブジェクトを作成する準備ができました。収益、支出、利益の3つの系列を作成します。

```
C#

// 収益の系列を追加
var ds = new DataSeries();
ds.Label = "収益";
ds.ValuesSource = (from r in data select r.Revenue).ToArray();
c1Chart.Data.Children.Add(ds);
// 支出の系列を追加
ds = new DataSeries();
ds.Label = "支出";
ds.ValuesSource = (from r in data select r.Expense).ToArray();
c1Chart.Data.Children.Add(ds);
// 利益の系列を追加
ds = new DataSeries();
ds.Label = "利益";
ds.ValuesSource = (from r in data select r.Profit).ToArray();
c1Chart.Data.Children.Add(ds);
```

コードでは、系列ごとに新しい **DataSeries** オブジェクトを作成して、その **Label** プロパティを設定しています。ラベルの設定は任意です。設定した場合は、このグラフに関連付けられたすべての **C1ChartLegend** オブジェクトに表示されます。次に、Linq ステートメントを使用して、データソースから値を取得します。その結果は、データ系列のオブジェクトの **ValuesSource** プロパティに割り当てられます。最後に、データソースはグラフの **Children** コレクションに追加されます。

ここでも、Linq の使用によってコードがいかに簡潔かつ自然なものになるかに注意してください。

手順4:ラフの外観の調整

Theme プロパティを使用して、グラフの外観を手早く設定します。

WPF

```
C#

// テーマを設定
c1Chart.Theme = _c1Chart.TryFindResource(new
ComponentResourceKey(typeof(C1.WPF.C1Chart.C1Chart), "Office2007Black")) as
ResourceDictionary;
```

Silverlight

```
C#

// テーマを設定します
c1Chart.Theme = c1Chart.TryFindResource(new
ComponentResourceKey(typeof(C1.Silverlight.C1Chart.C1Chart), "Office2007Black")) as
```

```
ResourceDictionary;}
}
```

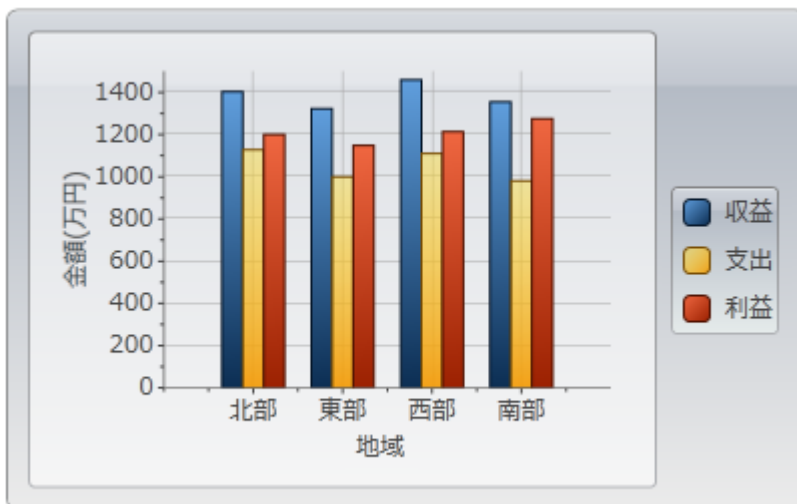
軸を設定するときに **CreateTextBlock** ヘルパーメソッドを使用したことを思い出してください。このメソッドの定義は次のとおりです。

C#

```
TextBlock CreateTextBlock(string text, double fontSize, FontWeight fontWeight)
{
    var tb = new TextBlock();
    tb.Text = text;
    tb.FontSize = fontSize;
    tb.FontWeight = fontWeight;
    return tb;
}
```

これで、単純な値のグラフを生成するコードは終わりです。このコードは、**ChartType** プロパティの値を他の任意の単純なグラフタイプの値 (Bar、AreaStacked、Pie) に変更して別タイプのグラフを作成することによってテストできます。**ChartType** を Column に変更する場合は、Y 軸にラベルを表示する必要があるため、AxisY を使用することに注意してください。結果は、以下の図のようになるはずです。

ChartType.Column



ChartType.Bar

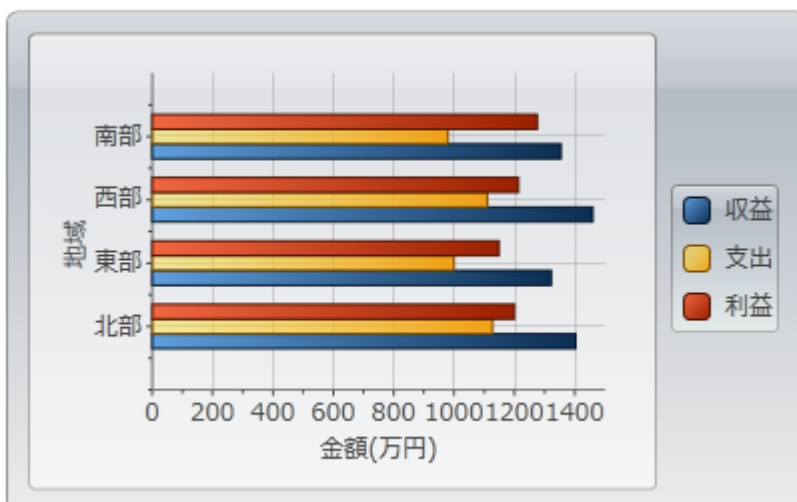
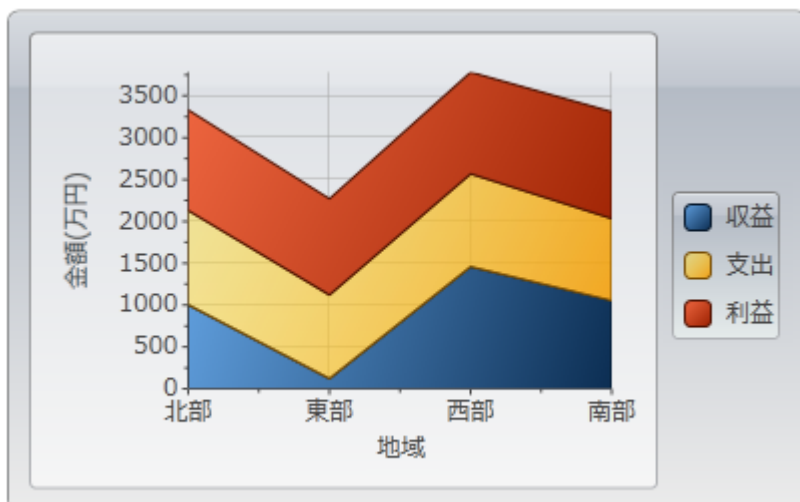
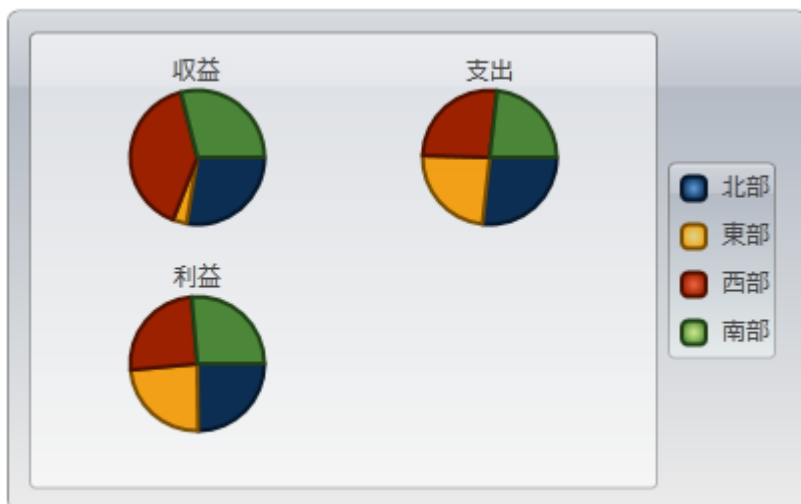


Chart for WPF/Silverlight

ChartType.AreaStacked



ChartType.Pie



<c1chart:C1ChartLegend DockPanel.Dock="Right" />

注意: デフォルトでは、系列を説明する凡例がグラフに表示されます。**C1ChartLegend** を削除するには、次の XAML コードを削除します。

特別なチャートタイプと複合チャート

XAML マークアップまたはコードを使用して、チャートの組み合わせを作成できます。これにより、C1Chart アプリケーションの柔軟性がさらに高まります。

棒系列と折れ線系列の追加

棒系列と折れ線系列をプログラムで追加するには、次のコードを使用します。

```
C#
chart.Data.Children.Add(new XYDataSeries() {
    ChartType=ChartType.Column,
    XValuesSource = new double[] {1,2,3 },
```



```

    ValuesSource = new double[] { 1, 2, 3 } });
chart.Children.Add(new XYDataSeries() {
    ChartType = ChartType.Line,
    XValuesSource = new double[] { 1, 2, 3 },
    ValuesSource = new double[] { 3, 2, 1 } });

```

縦棒 - 折れ線グラフ

-異なるデータ系列ごとに異なるテンプレートを使用することで、グラフタイプのさまざまな組み合わせを作成することができます。
 DataService.ChartType を使用してグラフを作成できます。

C#

```

<clchart:C1Chart Name="c1chart1">
  <clchart:C1Chart.Data>
    <clchart:ChartData >
      <!-- 最初目の系列のデフォルト(カラム)外観 -->
      <clchart:DataSeries Label="series 1" Values="0.5 2 3 4" />
      <!-- 二番目の系列は線につながる-->
      <clchart:DataSeries Label="series 2" Values="1 3 2 1"
        ChartType="LineSymbols" SymbolMarker="Star4" />
    </clchart:ChartData>
  </clchart:C1Chart.Data>
</clchart:C1Chart>

```

ガウス曲線の作成

ガウス曲線(正規曲線)は、ランダムな変数値の確率分布を示すために使用されます。

C1Chartでガウス曲線を作成するには、次のコードを使用します。

C#

```

// グラフデータ系列に作成して追加します
//   y(x) = a * exp( -(x-b)*(x-b) / (2*c*c) )
// x1 から x2 までの間隔で
void CreateGaussian(double x1, double x2, double a, double b, double c)
{
  // ポイントの数
  int cnt = 200;      var xvals = new double[cnt];
  var yvals = new double[cnt];
  double dx = (x2 - x1) / (cnt-1);
  for (int i = 0; i < cnt; i++)
  {
    var x = x1 + dx * i;
    xvals[i] = x;
    x = (x - b) / c;
    yvals[i] = a * Math.Exp(-0.5*x*x);
  }
  var ds = new XYDataSeries()
  {
    XValuesSource = xvals,

```

```
        ValuesSource = yvals,  
        ChartType = ChartType.Line  
    };  
    chart.Data.Children.Add(ds);  
}
```

パレート図の作成

パレート図は、大規模なデータセット内の最も重要な要素を強調します。

パレート図を作成するには、次の XAML マークアップを使用します。

C#

```
<clchart:C1Chart Name="c1Chart1">  
    <clchart:C1Chart.View>  
        <clchart:ChartView>  
            <clchart:ChartView.AxisX>  
                <clchart:Axis AnnoAngle="-75" MajorGridStroke="Gray"/>  
            </clchart:ChartView.AxisX>  
            <!-- 左の標準(デフォルト)y 軸 -->  
            <clchart:ChartView.AxisY>  
                <clchart:Axis Min="0" Max="50" Title="{StaticResource ytitle}"  
MajorGridStroke="Gray"/>  
            </clchart:ChartView.AxisY>  
            <!-- 補助(右)の y 軸 -->  
            <clchart:Axis Name="ay2" AxisType="Y" Position="Far" AnnoFormat="p"  
                Min="0" Max="1" />  
            </clchart:ChartView>  
        </clchart:C1Chart.View>  
        <clchart:C1Chart.Data>  
            <clchart:ChartData>  
                <clchart:ChartData.ItemNames>Documents Quality Packaging Delivery  
Other</clchart:ChartData.ItemNames>  
                <clchart:DataSeries Values="40 30 20 5 5" />  
                <clchart:DataSeries AxisY="ay2" Values="0.4 0.7 0.9 0.95 1.0"  
ChartType="LineSymbols" />  
            </clchart:ChartData>  
        </clchart:C1Chart.Data>  
    </clchart:C1Chart>
```

グラフ機能

ここでは、Chart for WPF/Silverlight コントロールを初めて使用するためのトピックを取り上げます。これらのトピックを利用して、チャートの外観からデータ連結まで、C1Chart のさまざまな部分をカスタマイズできます。

アニメーション

組み込みのアニメーション API を使用することで、ほぼすべてのプロット要素をアニメーションできます。組み込みのアニメーションオプションを使用して、**C1Chart** コントロールのプロット要素に、さまざまな視覚的なアニメーション効果を簡単に作成できます。**PlotElementAnimation** クラスに含まれるプロパティは、次のとおりです。

| プロパティ | 説明 |
|--|---|
| PlotElementAnimation.IndexDelay Attached | 要素のポイントインデックスに基づいてアニメーションの遅延を指定できる添付プロパティ。 |
| PlotElementAnimation.Storyboard | プロット要素に適用されるストーリーボードを取得または設定します。 |
| PlotElementAnimation.SymbolStyle | ストーリーボードが開始する前にプロット要素に適用されるシンボルスタイルを取得または設定します。 |

C1Chart コントロールのアニメーション表示には、**LoadAnimation** も使用されます。

組み込みのアニメーションオプションを使用してアニメーション効果を設定するために、次の XAML マークアップを使用できます。

```
<clchart:C1Chart Name="chart">
  <clchart:C1Chart.Data>
    <clchart:ChartData>
      <clchart:ChartData.LoadAnimation>
        <!-- アニメーションをロードします -->
        <clchart:PlotElementAnimation>
          <!-- 初期スタイル:非表示 -->
          <clchart:PlotElementAnimation.SymbolStyle>
            <Style TargetType="clchart:PlotElement">
              <Setter Property="Opacity" Value="0" />
            </Style>
          </clchart:PlotElementAnimation.SymbolStyle>
          <clchart:PlotElementAnimation.Storyboard>
            <Storyboard >
              <!-- インデックスディレイを使用して要素を表示します -->
              <DoubleAnimation
                Storyboard.TargetProperty="Opacity"
                clchart:PlotElementAnimation.IndexDelay="0.5"
                To="1" Duration="0:0:1" />
            </Storyboard>
          </clchart:PlotElementAnimation.Storyboard>
        </clchart:PlotElementAnimation>
      </clchart:ChartData.LoadAnimation>
    </clchart:ChartData>
  </clchart:C1Chart.Data>
</clchart:C1Chart>
```

次に、**InitializeComponent()** メソッドのすぐ下に次のコードを挿入します。

```
var rnd = new Random();
chart.MouseLeftButtonDown += (s, e) =>
{
    chart.Data.Children.Clear();
    // 新しいデータを作成します
    var vals = new double[rnd.Next(5, 10)];
    for (int i = 0; i < vals.Length; i++)
        vals[i] = rnd.Next(0, 100);
    chart.Data.Children.Add(new DataSeries() { ValuesSource = vals });
};
```

これで、アプリケーションを実行すると、データはマウスクリックでリロードされ、上記のマークアップで設定されたアニメーション効果を表示します。

カスタムアニメーションの作成

プロット要素は、ほとんどすべて標準の WPF/Silverlight アニメーションでアニメ化できます。次のスタイルは、「走るアリ」のアニメーションをマウスポインタの下にある要素に追加する修正版のスタイルです。

```
<Style x:Key="mouseOver" TargetType="{x:Type clc:PlotElement}">
  <!-- デフォルトの黒色の輪郭 -->
  <Setter Property="Stroke" Value="Black" />
  <Style.Triggers>
  <!-- マウスポインタが要素上にあるときに太い赤色の輪郭を作成 -->
    <Trigger Property="IsMouseOver" Value="true">
      <Setter Property="Stroke" Value="Red" />
      <Setter Property="StrokeThickness" Value="2" />
      <Setter Property="StrokeDashArray" Value="2,2" />
      <Setter Property="Canvas.ZIndex" Value="1" />
      <Trigger.EnterActions>
        <!-- アニメーションを開始 -->
        <BeginStoryboard >
          <Storyboard>
            <DoubleAnimation
Storyboard.TargetProperty="StrokeDashOffset"
              From="0" To="8" RepeatBehavior="Forever"
Duration="0:0:0.5"/>
          </Storyboard>
        </BeginStoryboard>
      </Trigger.EnterActions>
    </Trigger>
  </Style.Triggers>
</Style>
```

グラフの各 **DataSeries** は、系列内の個々の記号、コネクタ、領域、円のスライスなどを表す **PlotElement** オブジェクトで構成されます。**PlotElement** の具体的な型は、グラフタイプによって決まります。

アニメーションをグラフに追加するには、**Storyboard** オブジェクトをプロット要素に関連付けます。これは通常、**DataSeries.Loaded** イベントに反応して行います。このイベントは、**PlotElement** オブジェクトが作成されてデータ系列に

追加された後に発生します

カスタムアニメーションの作成

プロット要素は、ほとんどすべて標準の WPF/Silverlight アニメーションでアニメ化できます。次のスタイルは、「走るアリ」のアニメーションをマウスポインタの下にある要素に追加する修正版のスタイルです。

```
<Style x:Key="mouseOver" TargetType="{x:Type clc:PlotElement}">
  <!-- デフォルトの黒色の輪郭 -->
  <Setter Property="Stroke" Value="Black" />
  <Style.Triggers>
    <!-- マウスポインタが要素上にあるときに太い赤色の輪郭を作成 -->
    <Trigger Property="IsMouseOver" Value="true">
      <Setter Property="Stroke" Value="Red" />
      <Setter Property="StrokeThickness" Value="2" />
      <Setter Property="StrokeDashArray" Value="2,2" />
      <Setter Property="Canvas.ZIndex" Value="1" />
      <Trigger.EnterActions>
        <!-- アニメーションを開始 -->
        <BeginStoryboard >
          <Storyboard>
            <DoubleAnimation
Storyboard.TargetProperty="StrokeDashOffset"
              From="0" To="8" RepeatBehavior="Forever"
Duration="0:0:0.5"/>
          </Storyboard>
        </BeginStoryboard>
      </Trigger.EnterActions>
    </Trigger>
  </Style.Triggers>
</Style>
```

グラフの各 **DataSeries** は、系列内の個々の記号、コネクタ、領域、円のスライスなどを表す **PlotElement** オブジェクトで構成されます。**PlotElement** の具体的な型は、グラフタイプによって決まります。

アニメーションをグラフに追加するには、**Storyboard** オブジェクトをプロット要素に関連付けます。これは通常、**DataSeries.Loaded** イベントに反応して行います。このイベントは、**PlotElement** オブジェクトが作成されてデータ系列に追加された後に発生します

軸

軸は、**View** プロパティのサブプロパティである **AxisX**、**AxisY**、および **AxisZ** で表されます。持つ **Axis** オブジェクトを返します。

持つ **Axis** オブジェクトを返します。

レイアウト、スタイル、および値のプロパティ

以下のプロパティは、**C1Chart** の軸のレイアウトとスタイルを表します。

| プロパティ | 説明 |
|----------|---|
| Position | 軸の位置を設定できます。たとえば、X 軸をデータの下ではなく上に表示できます。 |

Chart for WPF/Silverlight

| | |
|------------|--|
| | 詳細については、「 軸の位置 」を参照してください。 |
| Reversed | 軸の方向を反転できます。たとえば、下から上ではなく上から下に向かって Y の値を表示できます。詳細については、「 グラフの軸の反転と逆転 」を参照してください。 |
| Title | 軸の隣に表示する文字列を設定します(これは通常、その軸で表現される変数と単位の説明に使用します)。詳細については、「 軸のタイトル 」を参照してください。 |
| Foreground | 軸の前景ブラシを取得または設定します。 |
| AxisLine | 軸の線を取得または設定します。軸の線は、軸の最小値と最大値に対応するプロット上のポイントを接続します。 |
| IsTime | 軸が時間の値を表すかどうかを取得または設定します。 |
| Scale | 軸のスケールを取得または設定します。 |
| MinScale | 軸の最小スケールを取得または設定します。 |

注釈のプロパティ

以下のプロパティは、**C1Chart** の注釈の書式を表します。

| プロパティ | 説明 |
|--------------|---|
| ItemsSource | 軸の注釈のソースを取得または設定します。 |
| AnnoFormat | 軸の隣に表示される値の書式設定に使用される一連の定義済み書式。 |
| AnnoAngle | 値を回転させることで、軸近傍の占有スペースを小さくできます。詳細については、「 軸の注釈の回転 」を参照してください。 |
| AnnoTemplate | 軸の注釈のテンプレートを取得または設定します。 |

スケール調整、目盛記号、およびグリッド線のプロパティ

以下のプロパティは、**C1Chart** の軸について、スケール調整、目盛記号、およびグリッド線のスタイルと機能を表します。

| プロパティ | 説明 |
|---|---|
| AutoMin、AutoMax | 軸の最小値と最大値を自動的に計算するかどうかを決定します。詳細については、「 軸の範囲 」を参照してください。 |
| Min、Max | 軸の最小値と最大値を設定します (AutoMin と AutoMax が False に設定されているとき)。詳細については、「 軸の範囲 」を参照してください。 |
| MajorUnit、MinorUnit | 主目盛記号と補助目盛記号の間のスペースを設定します (AutoMajor プロパティと AutoMinor プロパティが False に設定されているとき)。 |
| MajorGridFill | 主グリッドの塗りつぶしを取得または設定します。MajorGridFill では、プロットの外観を縞模様に変えます。 |
| MajorGridStroke、MinorGridStroke | 主グリッド線と副グリッド線のブラシを取得または設定します。 |
| MajorGridStrokeDashes、MinorGridStrokeDashes | 主グリッド線と副グリッド線のダッシュパターンを取得または設定します。 |

| | |
|---|------------------------------|
| MajorGridStrokeThickness、 MinorGridStrokeThickness | 主グリッド線と副グリッド線の太さを取得または設定します。 |
| MajorTickHeight、 MinorTickHeight | 主目盛と補助目盛の高さを取得または設定します。 |
| MajorTickStroke、 MinorTickStroke | 主目盛と補助目盛のストロークを取得または設定します。 |
| MajorTickThickness、 MinorTickThickness | 主目盛と補助目盛の太さを取得または設定します。 |

[先頭に戻る](#)

軸の注釈

各軸の注釈は、どのようなグラフでも重要な部分です。グラフでは、**BubbleSeries**、**DataSeries**、**HighLowOpenCloseSeries**、**HighLowSeries**、**XYDataSeries** といったオブジェクトに入力されたデータや値に基づいた数値で、軸に注釈が付けられます。軸の注釈には、常に書式設定が適用されていない基本的なテキストが表示されます。

グラフは、データが変化しても可能なかぎり自然な注釈を自動生成します。以下の注釈のプロパティを変更すれば、このプロセスを改良できます。

| プロパティ | 説明 |
|---------------------|--|
| AnnoFormat | 軸の隣に表示される値の書式設定に使用される一連の定義済み書式。 |
| AnnoAngle | 軸の注釈の回転角を取得または設定します。これにより値を回転させることで、軸近傍の占有スペースを小さくできます。 |
| AnnoTemplate | 軸の注釈のテンプレートを取得または設定します。これは、カスタム注釈を作成するときに便利です。例については、「 カスタム注釈の作成 」を参照してください。 |
| ItemsSource | 軸の注釈のソースを取得または設定します。 |

軸の注釈の書式

X 軸または Y 軸の値に関する注釈の書式設定は、**AnnoFormat** プロパティを使用して制御できます。

AnnoFormat プロパティを .NET Framework の複合書式文字列に設定すると、そのプロパティに入力されたデータが書式設定されます。**AnnoFormat** プロパティに使用できる標準数値書式文字列の詳細については、「[標準数値書式文字列](#)」を参照してください。

日時書式文字列

日時書式文字列は、次の2つのカテゴリに分けられます。

- [標準日時書式文字列](#)
- [カスタム日時書式文字列](#)

数値書式文字列

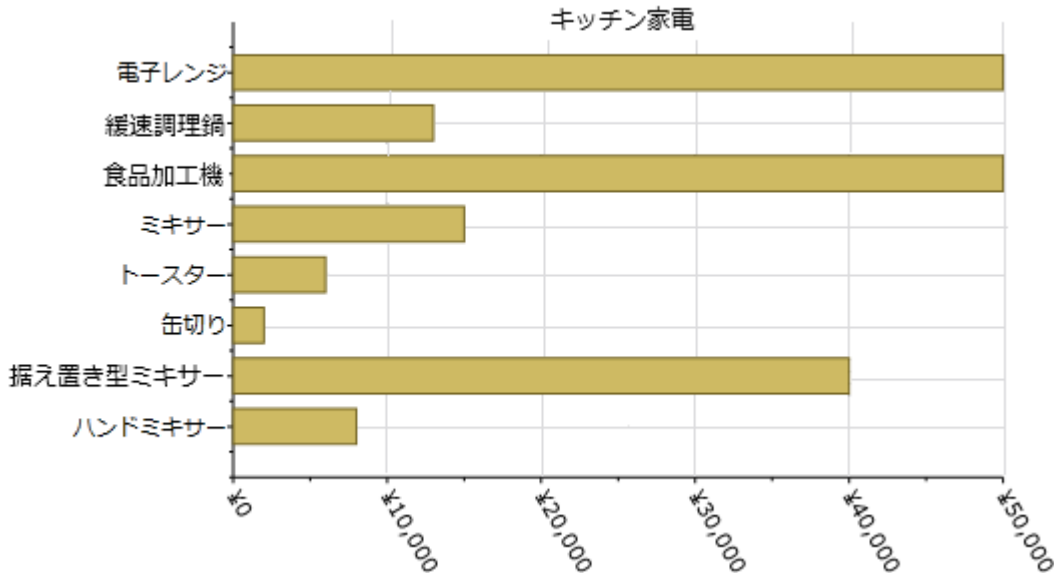
- [標準数値書式文字列](#)
- [カスタム数値書式文字列](#)

Chart for WPF/Silverlight

カスタム数値書式文字列

書式文字列は、カスタム数値書式文字列を使用してカスタマイズすることもできます。

AnnoFormat プロパティを使用するには、そのプロパティに対して標準またはカスタムの書式文字列を指定するだけです。たとえば、次の横棒グラフの **AnnoFormat** プロパティは、すべての値を通貨書式に変更するため、**c** に設定されています。



XAML

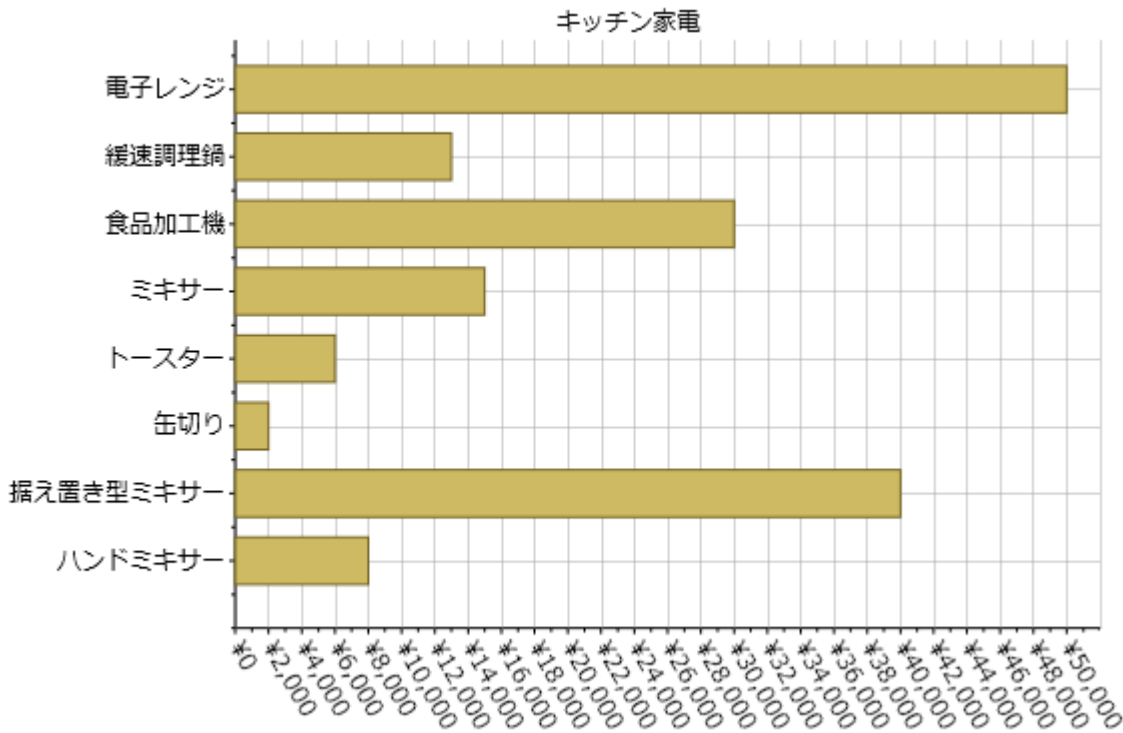
```
<clchart:C1Chart.View>
    <clchart:ChartView>
        <clchart:ChartView.AxisX>
            <clchart:Axis Min="0" AnnoFormat="c" AutoMin="false"
AutoMax="false" Max="50000" />
        </clchart:ChartView.AxisX>
    </clchart:ChartView>
</clchart:C1Chart.View>
```

C#

```
// 金融向けの書式設定
clChart1.View.AxisX.AnnoFormat = "c";
clChart1.View.AxisX.Min = 0;
```

軸の注釈の回転

軸の注釈を指定の角度から反時計回りに回転させるには、**AnnoAngle** プロパティを使用します。このプロパティは、X 軸に多数の注釈がある場合に特に有効です。注釈を +/- 30° または 60° 回転させると、水平軸近傍の限られたスペースに、はるかに多くの注釈を設定できます。**AnnoAngle** プロパティを利用すれば、下に示すとおり、X 軸の注釈が重なりません。



XAML

```
<clchart:C1Chart.View>
    <clchart:ChartView>
        <clchart:ChartView.AxisX>
            <clchart:Axis Min="0" MajorUnit="2000" AnnoFormat="c"
AutoMin="false" AutoMax="false" Max="50000" AnnoAngle="60" />
        </clchart:ChartView.AxisX>
    </clchart:ChartView>
</clchart:C1Chart.View>
```

C#

```
// 金融向けの書式設定
clChart1.View.AxisX.AnnoFormat = "c";
clChart1.View.AxisX.Min = 0;
clChart1.View.AxisX.AnnoAngle = "60";
```

軸のカスタム注釈

状況によっては、軸のカスタム注釈の作成が必要になる場合もあります。以下のシナリオは、軸のカスタム注釈を作成するために役立つ場合があります。

- **ItemsSource**プロパティが数値または **DateTime** 値のコレクションである場合、グラフでは、これらの値を軸のラベルとして使用します。次のコードは、**ItemsSource**プロパティを使用して、Y 軸のカスタムラベルを作成します。

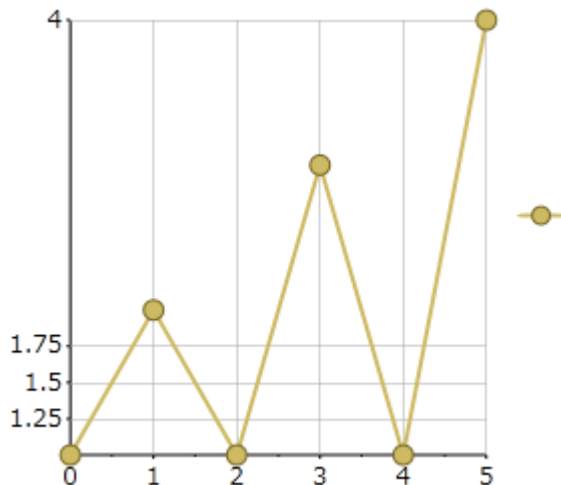
```
C#
```

Chart for WPF/Silverlight

```
c1Chart1.Reset(true);
    c1Chart1.Data.Children.Add(
        new DataSeries() { ValuesSource = new double[] { 1, 2, 1, 3, 1, 4 }
    });

    c1Chart1.ChartType = ChartType.LineSymbols;
    c1Chart1.View.AxisY.ItemsSource = new double[] { 1.25, 1.5, 1.75, 4 };
```

- 次の図は、上のコードを追加した後のグラフの表示です。

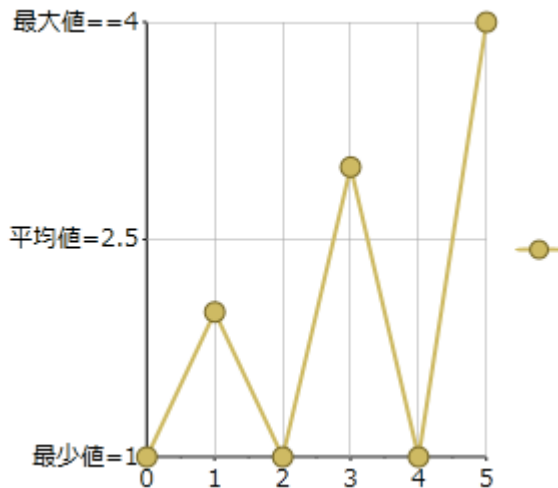


- **ItemsSource**プロパティが **KeyValuePair<object, double>** または **KeyValuePair<object, DateTime>** のコレクションである場合、グラフでは、指定された値のペアに基づいて軸のラベルが作成されます。たとえば、次のコードは、**KeyValuePair** を使用して Y 軸のカスタム注釈を作成します。

```
C#
c1Chart1.Reset(true);
    c1Chart1.Data.Children.Add(
        new DataSeries() { ValuesSource = new double[] { 1, 2, 1, 3, 1, 4 }
    });

    c1Chart1.ChartType = ChartType.LineSymbols;
    c1Chart1.View.AxisY.ItemsSource = new List<KeyValuePair<object, double>>
    {
        new KeyValuePair<object, double>("最小値=1", 1),
        new KeyValuePair<object, double>("平均値=2.5", 2.5),
        new KeyValuePair<object, double>("最大値=4", 4);
    };
```

- 次の図は、上のコードを追加した後のグラフの表示です。



- 次のコードのように、任意のコレクションをデータソースとして使用し、**ItemsValueBinding**プロパティと**ItemsLabelBinding**プロパティを使用して軸のラベルを作成できます。

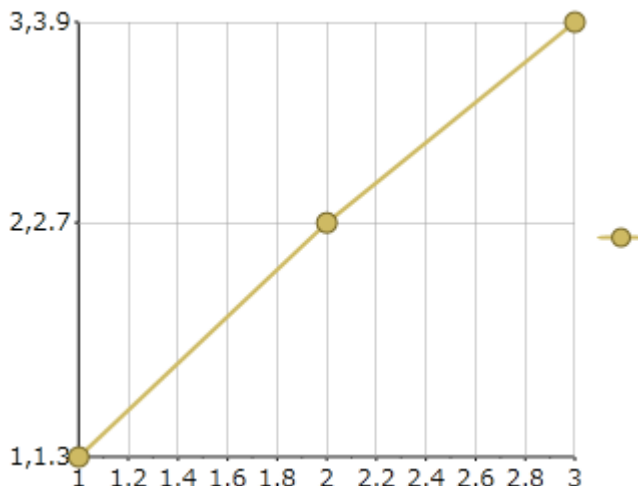
C#

```

c1Chart1.Reset(true);
    Point[] pts = new Point[] { new Point(1, 1.3), new Point(2, 2.7), new
Point(3, 3.9) };
    c1Chart1.DataContext = pts;
    c1Chart1.ChartType = ChartType.LineSymbols;
    c1Chart1.View.AxisY.ItemsSource = pts;
    c1Chart1.View.AxisY.ItemsValueBinding = new Binding("Y");
    c1Chart1.View.AxisY.ItemsLabelBinding = new Binding();

```

次の図は、上のコードを追加した後のグラフの表示です。



カスタム注釈の作成

AnnoTemplate プロパティを使用してカスタム注釈を作成するには、次の XAML コードまたは C# コードを使用します。

XAML

Chart for WPF/Silverlight

```
<clchart:ChartView.AxisX>
  <clchart:Axis>
    <clchart:Axis.Resources >
      <local:ColorConverter x:Key="clrcnv" />
    </clchart:Axis.Resources>
    <clchart:Axis.AnnoTemplate>
      <DataTemplate>
        <TextBlock Width="25" TextAlignment="Center"
          Text="{Binding Path=Value}"
          Foreground="{Binding Converter={StaticResource clrcnv}}"/>
      </DataTemplate>
    </clchart:Axis.AnnoTemplate>
  </clchart:Axis>
</clchart:ChartView.AxisX>
```

C#

```
public class ColorConverter : IValueConverter {
    int cnt = 0;
    public object Convert(object value, Type targetType,
        object parameter, CultureInfo culture)
    {
        //DataPoint dpt = (DataPoint)value;
        // ブラシを交互に切り替え
        return cnt++ % 2 == 0 ? Brushes.Blue : Brushes.Red;
    }

    public object ConvertBack(object value, Type targetType,
        object parameter, CultureInfo culture)
    {
        return null;
    }
}
```

軸と目盛りをチャートの反対側に表示する

チャートの反対側に水平軸と目盛りを表示するには、次のようなコードを使用して、タイトルのみを設定した補助軸をチャートの最上部に配置します。

VisualBasic

```
clChart1.View.Axes.Add(New Axis() With { _
    Key .AxisType = AxisType.X, _
    Key .Position = AxisPosition.Far, _
    Key .ItemsSource = New String() {""}, _
    Key .Title = "軸のタイトル" _
})
```

C#

```
clChart1.View.Axes.Add(new Axis()  
{  
    AxisType = AxisType.X,  
    Position = AxisPosition.Far,  
    ItemsSource = new string[] { "" },  
    Title = "軸のタイトル",  
});
```

軸の注釈の書式

X 軸または Y 軸の値に関する注釈の書式設定は、**AnnoFormat** プロパティを使用して制御できます。

AnnoFormat プロパティを .NET Framework の複合書式文字列に設定すると、そのプロパティに入力されたデータが書式設定されます。**AnnoFormat** プロパティに使用できる標準数値書式文字列の詳細については、「[標準数値書式文字列](#)」を参照してください。

日時書式文字列

日時書式文字列は、次の2つのカテゴリに分けられます。

- [標準日時書式文字列](#)
- [カスタム日時書式文字列](#)

数値書式文字列

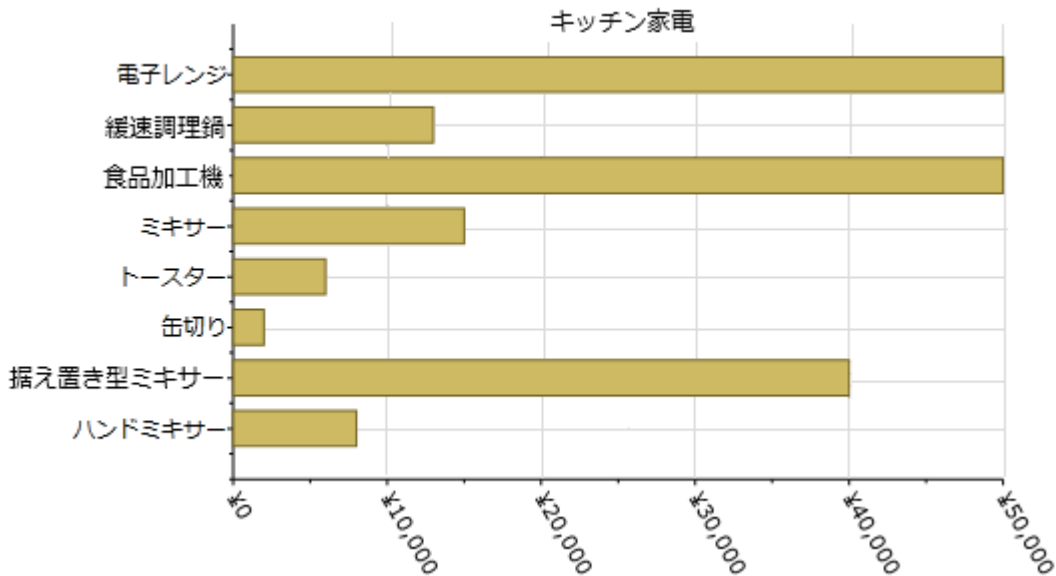
- [標準数値書式文字列](#)
- [カスタム数値書式文字列](#)

カスタム数値書式文字列

書式文字列は、カスタム数値書式文字列を使用してカスタマイズすることもできます。

AnnoFormat プロパティを使用するには、そのプロパティに対して標準またはカスタムの書式文字列を指定するだけです。たとえば、次の横棒グラフの **AnnoFormat** プロパティは、すべての値を通貨書式に変更するため、c に設定されています。

Chart for WPF/Silverlight



XAML

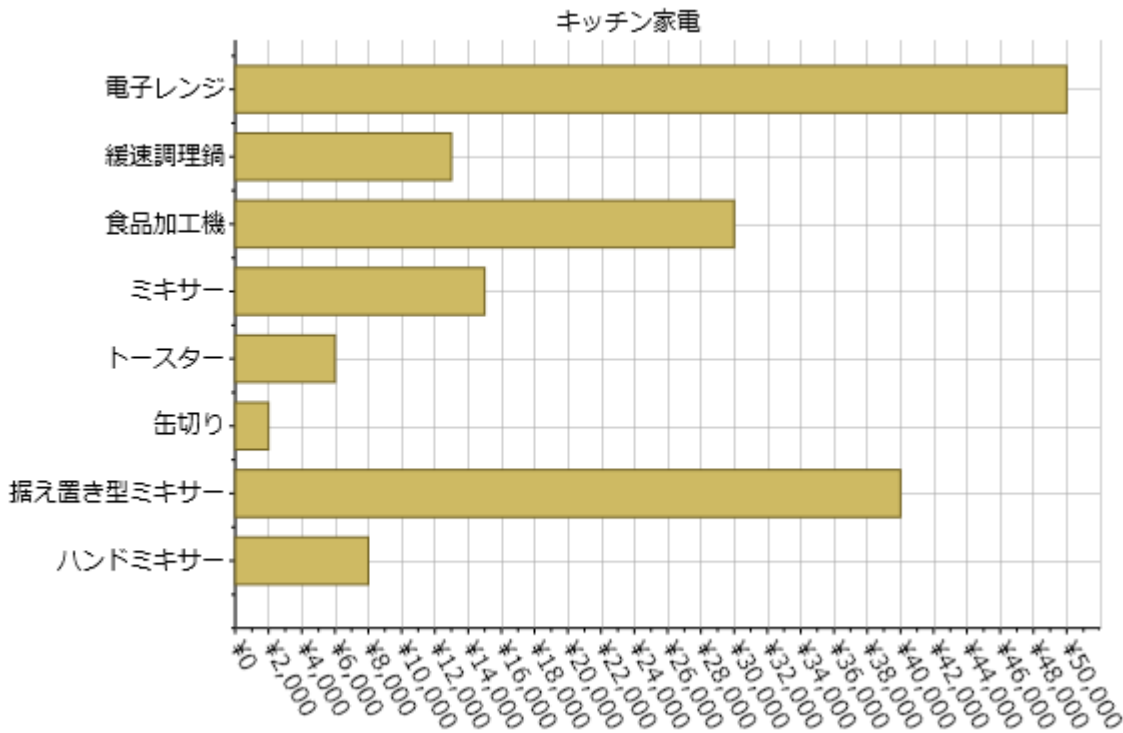
```
<clchart:C1Chart.View>
    <clchart:ChartView>
        <clchart:ChartView.AxisX>
            <clchart:Axis Min="0" AnnoFormat="c" AutoMin="false"
AutoMax="false" Max="50000" />
        </clchart:ChartView.AxisX>
    </clchart:ChartView>
</clchart:C1Chart.View>
```

C#

```
// 金融向けの書式設定
clChart1.View.AxisX.AnnoFormat = "c";
clChart1.View.AxisX.Min = 0;
```

軸の注釈の回転

軸の注釈を指定の角度から反時計回りに回転させるには、**AnnoAngle** プロパティを使用します。このプロパティは、X 軸に多数の注釈がある場合に特に有効です。注釈を +/- 30° または 60° 回転させると、水平軸近傍の限られたスペースに、はるかに多くの注釈を設定できます。**AnnoAngle** プロパティを利用すれば、下に示すとおり、X 軸の注釈が重なりません。



XAML

```
<clchart:C1Chart.View>
    <clchart:ChartView>
        <clchart:ChartView.AxisX>
            <clchart:Axis Min="0" MajorUnit="2000" AnnoFormat="c"
AutoMin="false" AutoMax="false" Max="50000" AnnoAngle="60" />
        </clchart:ChartView.AxisX>
    </clchart:ChartView>
</clchart:C1Chart.View>
```

C#

```
// 金融向けの書式設定
clChart1.View.AxisX.AnnoFormat = "c";
clChart1.View.AxisX.Min = 0;
clChart1.View.AxisX.AnnoAngle = "60";
```

軸のカスタム注釈

状況によっては、軸のカスタム注釈の作成が必要になる場合もあります。以下のシナリオは、軸のカスタム注釈を作成するために役立つ場合があります。

- **ItemsSource**プロパティが数値または **DateTime** 値のコレクションである場合、グラフでは、これらの値を軸のラベルとして使用します。次のコードは、**ItemsSource**プロパティを使用して、Y 軸のカスタムラベルを作成します。

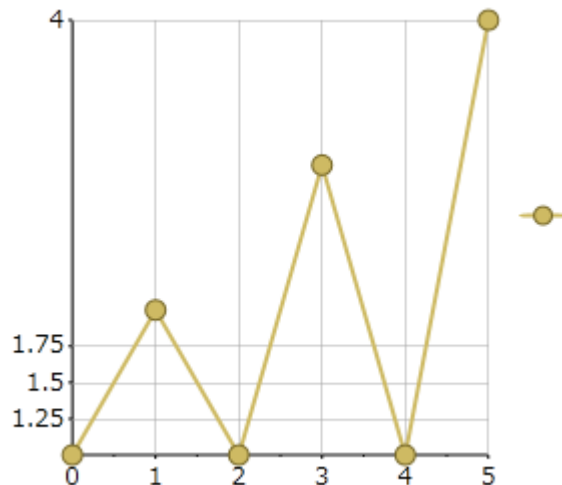
```
C#
```

Chart for WPF/Silverlight

```
c1Chart1.Reset(true);
    c1Chart1.Data.Children.Add(
        new DataSeries() { ValuesSource = new double[] { 1, 2, 1, 3, 1, 4 }
    });

    c1Chart1.ChartType = ChartType.LineSymbols;
    c1Chart1.View.AxisY.ItemsSource = new double[] { 1.25, 1.5, 1.75, 4 };
```

- 次の図は、上のコードを追加した後のグラフの表示です。

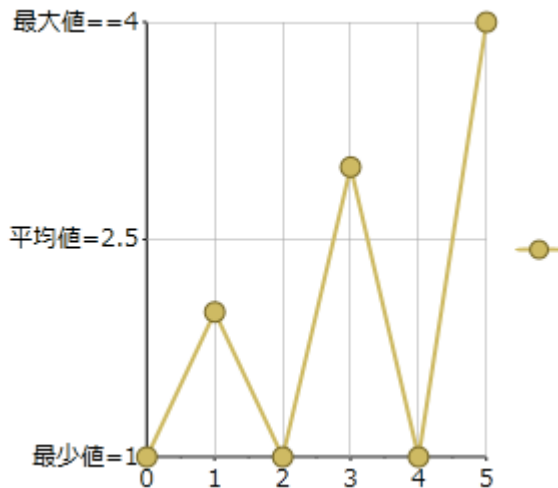


- **ItemsSource**プロパティが **KeyValuePair<object, double>** または **KeyValuePair<object, DateTime>** のコレクションである場合、グラフでは、指定された値のペアに基づいて軸のラベルが作成されます。たとえば、次のコードは、**KeyValuePair** を使用して Y 軸のカスタム注釈を作成します。

```
C#
c1Chart1.Reset(true);
    c1Chart1.Data.Children.Add(
        new DataSeries() { ValuesSource = new double[] { 1, 2, 1, 3, 1, 4 }
    });

    c1Chart1.ChartType = ChartType.LineSymbols;
    c1Chart1.View.AxisY.ItemsSource = new List<KeyValuePair<object, double>>
    {
        new KeyValuePair<object, double>("最小値=1", 1),
        new KeyValuePair<object, double>("平均値=2.5", 2.5),
        new KeyValuePair<object, double>("最大値=4", 4);
    };
```

- 次の図は、上のコードを追加した後のグラフの表示です。



- 次のコードのように、任意のコレクションをデータソースとして使用し、**ItemsValueBinding**プロパティと**ItemsLabelBinding**プロパティを使用して軸のラベルを作成できます。

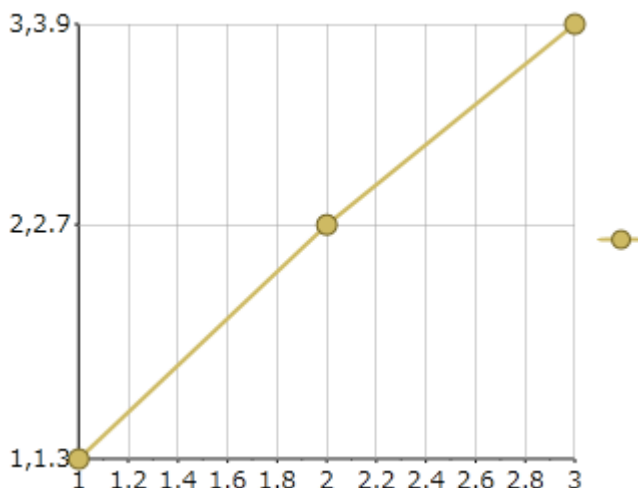
C#

```

c1Chart1.Reset(true);
    Point[] pts = new Point[] { new Point(1, 1.3), new Point(2, 2.7), new
Point(3, 3.9) };
    c1Chart1.DataContext = pts;
    c1Chart1.ChartType = ChartType.LineSymbols;
    c1Chart1.View.AxisY.ItemsSource = pts;
    c1Chart1.View.AxisY.ItemsValueBinding = new Binding("Y");
    c1Chart1.View.AxisY.ItemsLabelBinding = new Binding();

```

次の図は、上のコードを追加した後のグラフの表示です。



カスタム注釈の作成

AnnoTemplate プロパティを使用してカスタム注釈を作成するには、次の XAML コードまたは C# コードを使用します。

XAML

Chart for WPF/Silverlight

```
<clchart:ChartView.AxisX>
  <clchart:Axis>
    <clchart:Axis.Resources >
      <local:ColorConverter x:Key="clrcnv" />
    </clchart:Axis.Resources>
    <clchart:Axis.AnnoTemplate>
      <DataTemplate>
        <TextBlock Width="25" TextAlignment="Center"
          Text="{Binding Path=Value}"
          Foreground="{Binding Converter={StaticResource clrcnv}}"/>
      </DataTemplate>
    </clchart:Axis.AnnoTemplate>
  </clchart:Axis>
</clchart:ChartView.AxisX>
```

C#

```
public class ColorConverter : IValueConverter {
    int cnt = 0;
    public object Convert(object value, Type targetType,
        object parameter, CultureInfo culture)
    {
        //DataPoint dpt = (DataPoint)value;
        // ブラシを交互に切り替え
        return cnt++ % 2 == 0 ? Brushes.Blue : Brushes.Red;
    }

    public object ConvertBack(object value, Type targetType,
        object parameter, CultureInfo culture)
    {
        return null;
    }
}
```

軸と目盛りをチャートの反対側に表示する

チャートの反対側に水平軸と目盛りを表示するには、次のようなコードを使用して、タイトルのみを設定した補助軸をチャートの最上部に配置します。

VisualBasic

```
clChart1.View.Axes.Add(New Axis() With { _
    Key .AxisType = AxisType.X, _
    Key .Position = AxisPosition.Far, _
    Key .ItemsSource = New String() {""}, _
    Key .Title = "軸のタイトル" _
})
```

C#

```

c1Chart1.View.Axes.Add(new Axis()
{
    AxisType = AxisType.X,
    Position = AxisPosition.Far,
    ItemsSource = new string[] { ""},
    Title = "軸のタイトル",
});

```

軸の線

軸の線は、Y 軸については開始値から終了値まで水平に表示され、X 軸については開始値から終了値まで垂直に表示されます。Z 軸は 3D グラフで使用されます。

軸の線に色を適用するには、**Axis.Foreground** プロパティまたは **ShapeStyle.Stroke** プロパティのいずれかを使用できます。**Axis.Foreground** プロパティは **ShapeStyle.Stroke** プロパティをオーバーライドすることに注意してください。

軸の線の始点と終点の形状のタイプは、**StrokeStartLineCap** プロパティと **StrokeEndLineCap** プロパティを使用して指定できます。

従属軸

新しいプロパティ **IsDependent** を使用すれば、補助の軸を主軸 (AxisType に応じて AxisX または AxisY) の 1 つにリンクできます。従属軸の最小値と最大値は、常に主軸と同じです。

新しいプロパティ **DependentAxisConverter** とデリゲートの **Axis.AxisConverter** は、主軸から従属軸への座標変換に使用される関数を規定します。

次のコードは、従属した Y 軸を作成します。

```

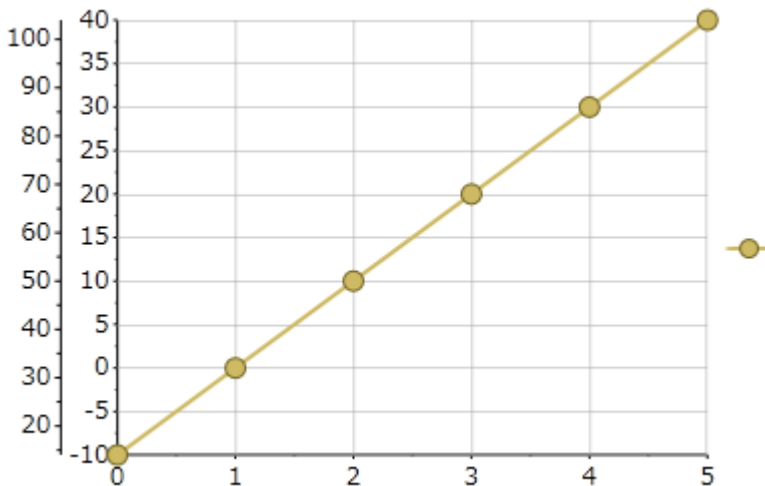
C#
c1Chart1.Reset(true);
c1Chart1.Data.Children.Add(
    new DataSeries() { ValuesSource = new double[] { -10, 0, 10, 20, 30, 40 }
});

c1Chart1.ChartType = ChartType.LineSymbols;
Axis axis = new Axis() { AxisType = AxisType.Y, IsDependent = true };
// 摂氏 -> 華氏
axis.DependentAxisConverter = (val) => val * 9 / 5 + 32;
c1Chart1.View.Axes.Add(axis);

```

次の図は、従属した Y 軸を示しています。

Chart for WPF/Silverlight



軸の位置

軸の位置は、**Position** プロパティを Near または Far の値に設定することによって指定できます。垂直の軸では、**Axis.Position.Near** は左に対応し、**Axis.Position.Far** は右に対応します。水平の軸では、**Axis.Position.Near** は下に対応し、**Axis.Position.Far** は上に対応します。

軸の範囲

通常、グラフでは、含まれているデータがすべて表示されます。しかし、軸の範囲を調整することによって、グラフの特定の部分を表示することもできます。

グラフでは、最低値と最高値、および数値の増分を考慮することによって、各軸の範囲が決まります。**Min** プロパティと**Max** プロパティ、**AutoMin** プロパティと**AutoMax** プロパティを設定すれば、このプロセスをカスタマイズできます。

軸の最小値と最大値

特定の軸の値でグラフの枠を設定するには、**Min** プロパティと**Max** プロパティを使用します。グラフの X 軸の値が 0 ~ 100 の場合、**Min** を 0、**Max** を 10 に設定すると、値は 10 までしか表示されません。

グラフでは、**Min** と**Max** の値を自動的に計算することもできます。**AutoMax** プロパティと**AutoMin** プロパティを **True** に設定した場合、グラフでは、現在のデータセットに合わせて軸の数値が自動設定されます。

軸の原点を設定する

軸の原点は、次のように**Origin** プロパティを使って指定できます。

C#

```
{
    c1Chart1.Reset(true);
    c1Chart1.Data.Children.Add(new XYDataSeries()
    {
        ValuesSource = new double[] { -1, 2, 0, 2, -2 },
        XValuesSource = new double[] { -2, -1, 0, 1, 2 }
    });
    c1Chart1.View.AxisX.Origin = 0;
    c1Chart1.View.AxisY.Origin = 0;
}
```

```

c1Chart1.ChartType = ChartType.LineSymbols;
});

```

軸のタイトル

軸にタイトルを追加すると、その軸で何が表示されているかが明確になります。たとえば、データに測定値が含まれている場合は、測定の単位(グラム、メートル、リットルなど)を軸のタイトルに含めると効果的です。軸のタイトルは、**エリアグラフ**、**XYプロットグラフ**、**横棒グラフ**、**HiLoOpenClose** グラフ、**ローソク足**グラフなどに追加できます。

軸のタイトルは単なるテキストではなく、UIElement オブジェクトです。これは、タイトルの書式を完全に制御できるということです。実は、軸のタイトルには、ボタン、テーブル、または画像を備えた複雑な要素を使用できます。

プログラムによる軸のタイトルの設定

C#

```

// 軸のタイトルを設定
c1Chart1.View.AxisY.Title= new TextBlock() { Text = "キッチン家電" };
c1Chart1.View.AxisX.Title = new TextBlock() { Text = "価格" };

```

XAML コードを使用した軸のタイトルの設定

XAML

```

<clchart:C1Chart >
  <clchart:C1Chart.View>
    <clchart:ChartView>
      <clchart:ChartView.AxisX>
        <clchart:Axis>
          <clchart:Axis.Title>
            <TextBlock Text="価格" TextAlignment="Center" Foreground="Crimson"/>
          </clchart:Axis.Title>
        </clchart:Axis>
      </clchart:ChartView.AxisX>
      <clchart:ChartView.AxisY>
        <clchart:Axis>
          <clchart:Axis.Title>
            <TextBlock Text="キッチン家電" TextAlignment="Center"
Foreground="Crimson"/>
          </clchart:Axis.Title>
        </clchart:Axis>
      </clchart:ChartView.AxisY>
    </clchart:ChartView>
  </clchart:C1Chart.View>
</clchart:C1Chart>

```

軸の目盛記号

グラフでは、軸は主目盛と補助目盛付きで自動的に設定されます。目盛の間隔や属性のカスタマイズは、一連のプロパティを操作するだけでできます。

MajorUnit プロパティと **MinorUnit** プロパティは、軸の目盛記号を設定します。グラフの乱雑さを解消するには、カテゴリのラベル付けの間隔を指定したり、目盛記号の間に表示するカテゴリの数を指定したりすることによって、カテゴリ(x)軸に表示

Chart for WPF/Silverlight

するラベルや目盛記号を削減できます。

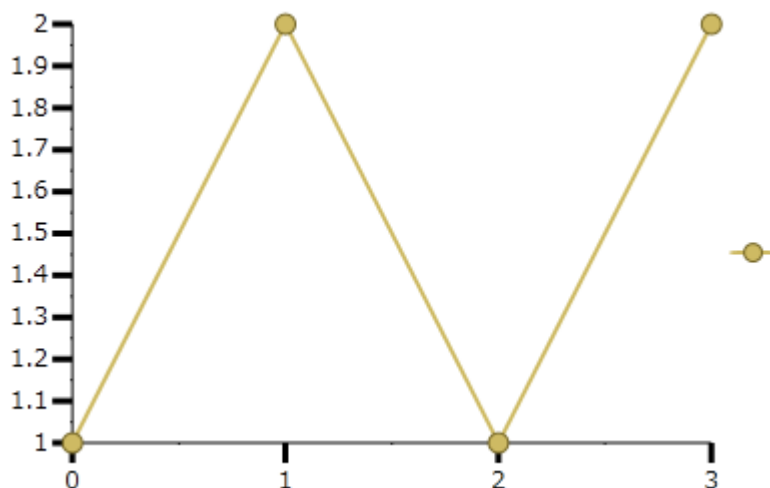
主目盛のオーバーラップ

MajorTickOverlap プロパティの値を0~1の範囲で指定することによって、主目盛記号のオーバーラップ値を決定できます。デフォルト値は0で、その場合、オーバーラップはありません。オーバーラップが1のときは、目盛全体がプロット領域内に表示されます。X軸の **MajorTickOverlap** の値を増加させると、目盛記号は上に移動し、減少させると下に移動します。Y軸の **MajorTickOverlap** の値を増加させると、目盛記号は左に移動します。

C#

```
c1Chart1.Reset(true);
c1Chart1.Data.Children.Add(
new DataSeries() { ValuesSource = new double[] { 1, 2, 1, 2 } });
c1Chart1.ChartType = ChartType.LineSymbols;
c1Chart1.View.AxisX.MajorGridStrokeThickness = 0;
c1Chart1.View.AxisX.MajorTickThickness = 3;
c1Chart1.View.AxisX.MajorTickHeight = 10;
c1Chart1.View.AxisX.MajorTickOverlap = 0;
c1Chart1.View.AxisY.MajorGridStrokeThickness = 0;
c1Chart1.View.AxisY.MajorTickThickness = 3;
c1Chart1.View.AxisY.MajorTickHeight = 10;
c1Chart1.View.AxisY.MajorTickOverlap = 0;
```

次の図は、**MajorTickOverlap** 値が0のときの表示です。



補助目盛のオーバーラップ

MinorTickOverlap プロパティの値を0~1の範囲で指定することによって、補助目盛記号のオーバーラップ値を決定できます。デフォルト値は0で、その場合、オーバーラップはありません。オーバーラップが1のときは、目盛全体がプロット領域内に表示されます。

C#

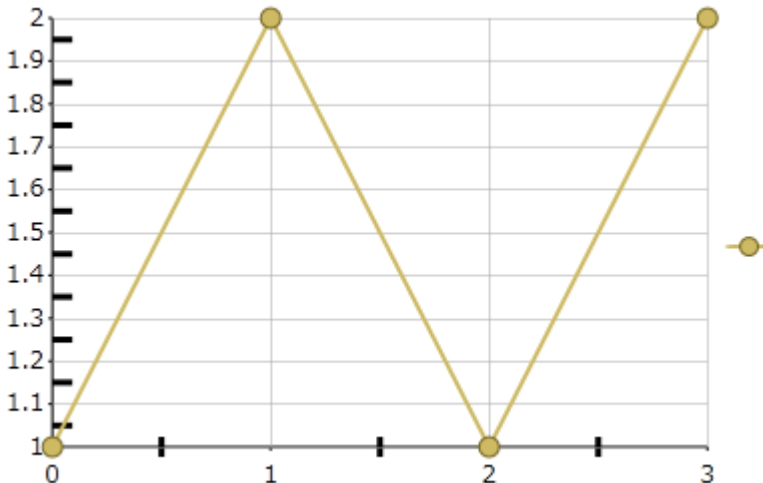
```
c1Chart1.Reset(true);
c1Chart1.Data.Children.Add(
new DataSeries() { ValuesSource = new double[] { 1, 2, 1, 2 } });
```

```

c1Chart1.ChartType = ChartType.LineSymbols;
c1Chart1.View.AxisX.MinorGridStrokeThickness = 0;
c1Chart1.View.AxisX.MinorTickThickness = 3;
c1Chart1.View.AxisX.MinorTickHeight = 10;
c1Chart1.View.AxisX.MinorTickOverlap = .5;
c1Chart1.View.AxisY.MinorGridStrokeThickness = 0;
c1Chart1.View.AxisY.MinorTickThickness = 3;
c1Chart1.View.AxisY.MinorTickHeight = 10;
c1Chart1.View.AxisY.MinorTickOverlap = 1;

```

次の図は、**MinorTickOverlap** が「1」に設定されているときの表示です。



主目盛と補助目盛の指定

軸の目盛は2種類あります。主目盛は短い線で、ラベルが対応しています。一方、補助目盛は軸全体にわたり線があるだけです。

デフォルトでは、目盛の間隔は自動的に計算されます。

特定の間隔を指定するには、**MajorUnit** プロパティと **MinorUnit** プロパティを使用します。

デフォルトの目盛

次の図は、デフォルトの目盛を表示しています。

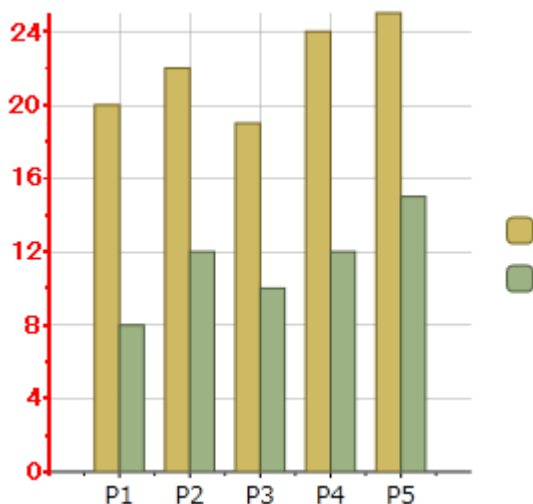


Chart for WPF/Silverlight

カスタム目盛

次のグラフ図では、**MajorUnit** プロパティと **MinorUnit** プロパティを使用して、特定の間隔を設定します。たとえば、次のとおりです。

VisualBasic

```
c1Chart1.View.AxisY.MajorUnit = 5  
c1Chart1.View.AxisY.MinorUnit = 1
```

C#

```
c1Chart1.View.AxisY.MajorUnit = 5;  
c1Chart1.View.AxisY.MinorUnit = 1;
```

時間軸

時間軸の場合は、**MajorUnit** と **MinorUnit** を `TimeSpan` の値として指定できます。

Visual Basic

```
c1Chart1.View.AxisY.MajorUnit = TimeSpan.FromHours(12)
```

C#

```
C#  
c1Chart1.View.AxisY.MajorUnit = TimeSpan.FromHours(12);
```

主目盛のオーバーラップ

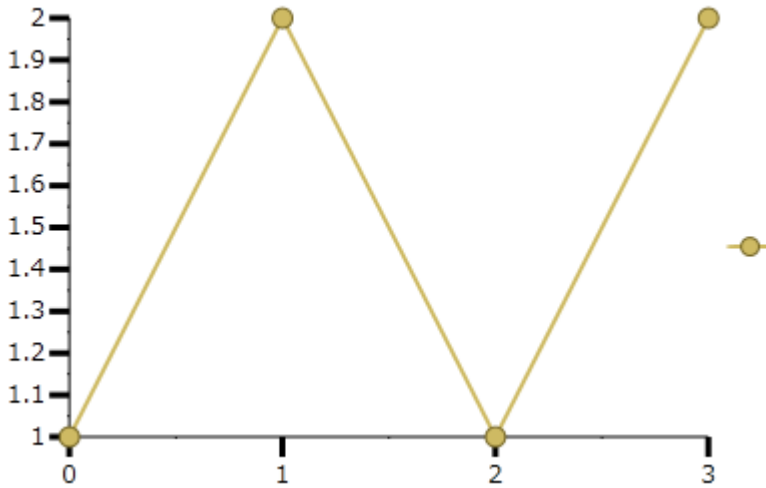
MajorTickOverlap プロパティの値を0~1の範囲で指定することによって、主目盛記号のオーバーラップ値を決定できます。デフォルト値は0で、その場合、オーバーラップはありません。オーバーラップが1のときは、目盛全体がプロット領域内に表示されます。X軸の **MajorTickOverlap** の値を増加させると、目盛記号は上に移動し、減少させると下に移動します。Y軸の **MajorTickOverlap** の値を増加させると、目盛記号は左に移動します。

```
C#  
c1Chart1.Reset(true);  
c1Chart1.Data.Children.Add(  
new DataSeries() { ValuesSource = new double[] { 1, 2, 1, 2 } });  
c1Chart1.ChartType = ChartType.LineSymbols;  
c1Chart1.View.AxisX.MajorGridStrokeThickness = 0;  
c1Chart1.View.AxisX.MajorTickThickness = 3;  
c1Chart1.View.AxisX.MajorTickHeight = 10;  
c1Chart1.View.AxisX.MajorTickOverlap = 0;  
c1Chart1.View.AxisY.MajorGridStrokeThickness = 0;  
c1Chart1.View.AxisY.MajorTickThickness = 3;
```



```
c1Chart1.View.AxisY.MajorTickHeight = 10;
c1Chart1.View.AxisY.MajorTickOverlap = 0;
```

次の図は、**MajorTickOverlap** 値が0のときの表示です。



補助目盛のオーバーラップ

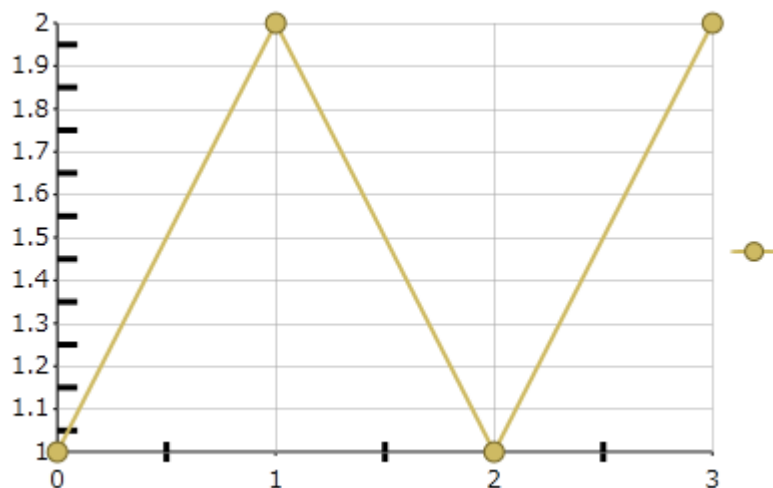
MinorTickOverlap プロパティの値を0~1の範囲で指定することによって、補助目盛りのオーバーラップ値を決定できます。デフォルト値は0で、その場合、オーバーラップはありません。オーバーラップが1のときは、目盛全体がプロット領域内に表示されます。

C#

```
c1Chart1.Reset(true);
c1Chart1.Data.Children.Add(
new DataSeries() { ValuesSource = new double[] { 1, 2, 1, 2 } });
c1Chart1.ChartType = ChartType.LineSymbols;
c1Chart1.View.AxisX.MinorGridStrokeThickness = 0;
c1Chart1.View.AxisX.MinorTickThickness = 3;
c1Chart1.View.AxisX.MinorTickHeight = 10;
c1Chart1.View.AxisX.MinorTickOverlap = .5;
c1Chart1.View.AxisY.MinorGridStrokeThickness = 0;
c1Chart1.View.AxisY.MinorTickThickness = 3;
c1Chart1.View.AxisY.MinorTickHeight = 10;
c1Chart1.View.AxisY.MinorTickOverlap = 1;
```

次の図は、**MinorTickOverlap** が「1」に設定されているときの表示です。

Chart for WPF/Silverlight



主目盛と補助目盛の指定

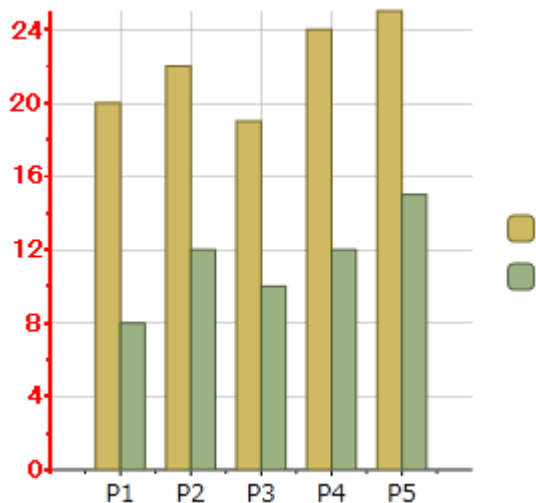
軸の目盛は2種類あります。主目盛は短い線で、ラベルが対応しています。一方、補助目盛は軸全体にわたり線があるだけです。

デフォルトでは、目盛の間隔は自動的に計算されます。

特定の間隔を指定するには、**MajorUnit** プロパティと **MinorUnit** プロパティを使用します。

デフォルトの目盛

次の図は、デフォルトの目盛を表示しています。



カスタム目盛

次のグラフ図では、**MajorUnit** プロパティと **MinorUnit** プロパティを使用して、特定の間隔を設定します。たとえば、次のとおりです。

VisualBasic

```
c1Chart1.View.AxisY.MajorUnit = 5  
c1Chart1.View.AxisY.MinorUnit = 1
```

C#

```
c1Chart1.View.AxisY.MajorUnit = 5;
c1Chart1.View.AxisY.MinorUnit = 1;
```

時間軸

時間軸の場合は、**MajorUnit** と **MinorUnit** を `TimeSpan` の値として指定できます。

Visual Basic

```
c1Chart1.View.AxisY.MajorUnit = TimeSpan.FromHours(12)
```

C#

```
C#
c1Chart1.View.AxisY.MajorUnit = TimeSpan.FromHours(12);
```

軸のグリッド線

グリッド線は、主／副単位間隔ごとに主目盛記号および補助目盛記号と直交して表示されます。グリッド線を設定すれば、正確な値を確認する際のグラフの読みやすさが向上します。

主／副グリッド線の色または塗りつぶしの設定

主グリッド線と副グリッド線の色は、**MajorGridStroke** プロパティと **MinorGridStroke** プロパティを使用して適用できます。塗りつぶし色は、**MajorGridFill** プロパティと **Axis.MinorGridFill** プロパティを使用して、主グリッド線または副グリッド線の各値の間で適用できます。

主／副グリッド線のダッシュパターンの設定

主グリッド線と副グリッド線のダッシュパターンは、**MajorGridStrokeDashes** プロパティと **MinorGridStrokeDashes** プロパティを使用して設定できます。

主／副グリッド線の太さの設定

主グリッド線と副グリッド線の太さは、**MajorGridStrokeThickness** プロパティと **MinorGridStrokeThickness** プロパティを使用して指定できます。

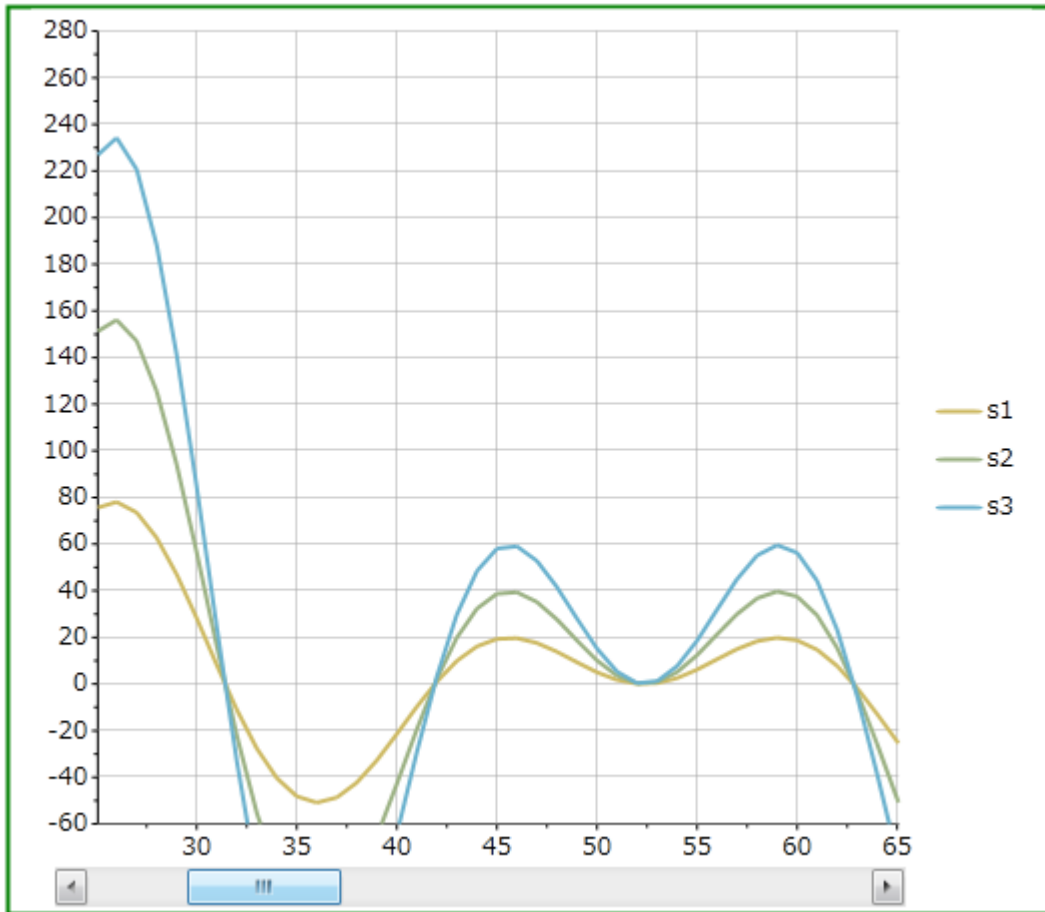
主グリッド線の塗りつぶしの設定

主グリッド線の塗りつぶしは、**MajorGridFill** プロパティを使用して適用できます。

軸のスクロール

グラフのデータで X 軸または Y 軸の値が相当な数にのぼる場合は、軸にスクロールバーを追加できます。スクロールバーを追加すると、スクロールによってデータを一部分ずつ詳細に表示できるため、グラフ上のデータの読み取りが容易になります。次の図では、スクロールバーを **View.AxisX.Value** プロパティに設定しています。

Chart for WPF/Silverlight



スクロールバーは、スクロールバーの **Value** プロパティを **AxisX**(X 軸の場合)または **AxisY**(Y 軸の場合)に設定するだけで、X 軸または Y 軸に表示できます。

次の XAML コードは、水平スクロールバーを X 軸に割り当てる方法を示しています。

XAML

```
<clchart:C1Chart Name="c1Chart1">
  <clchart:C1Chart.View>
    <clchart:ChartView>
      <clchart:ChartView.AxisX>
        <clchart:Axis Scale="0.2">
          <clchart:Axis.ScrollBar>
            <clchart:AxisScrollBar />
          </clchart:Axis.ScrollBar>
        </clchart:Axis>
      </clchart:ChartView.AxisX>
    </clchart:ChartView>
  </clchart:C1Chart.View>
```

スクロールバーの最小値と最大値を設定すると、スクロール時にスクロールバーによって軸の値が変化することはありません。

グラフの軸の反転と逆転

データセットにおいて X または Y の値が広範囲に及んでいる場合は、標準のグラフ設定では情報をあまり効果的に表示でき

ない場合があります。グラフを反転したり軸を逆転したりできる場合は、垂直の Y 軸と最小値で始まる軸の注釈を持つグラフを書式設定することで、視覚的なアピール力が増すことがあります。そのため、**C1Chart** では、**Inverted** プロパティと軸の **Reversed** プロパティが用意されています。

ChartView の **Reversed** プロパティを **True** に設定すると、軸が逆転します。これは、軸の Max 側が軸の Min 側の位置を占め、軸の Min 側が軸の Max 側の位置を占めるということです。当初、グラフでは X 軸の左端と Y 軸の下端に最小値が表示されます。しかし、軸の **Reversed** プロパティを設定すると、これらの最大値と最小値は並置されます。

X 軸と Y 軸を交換する

チャートを読み込んだ後で軸を反転するには、次のコードを使用します。

```
C#
clChart1.View.Inverted = true;
```

X 軸と Y 軸を交換する

チャートを読み込んだ後で軸を反転するには、次のコードを使用します。

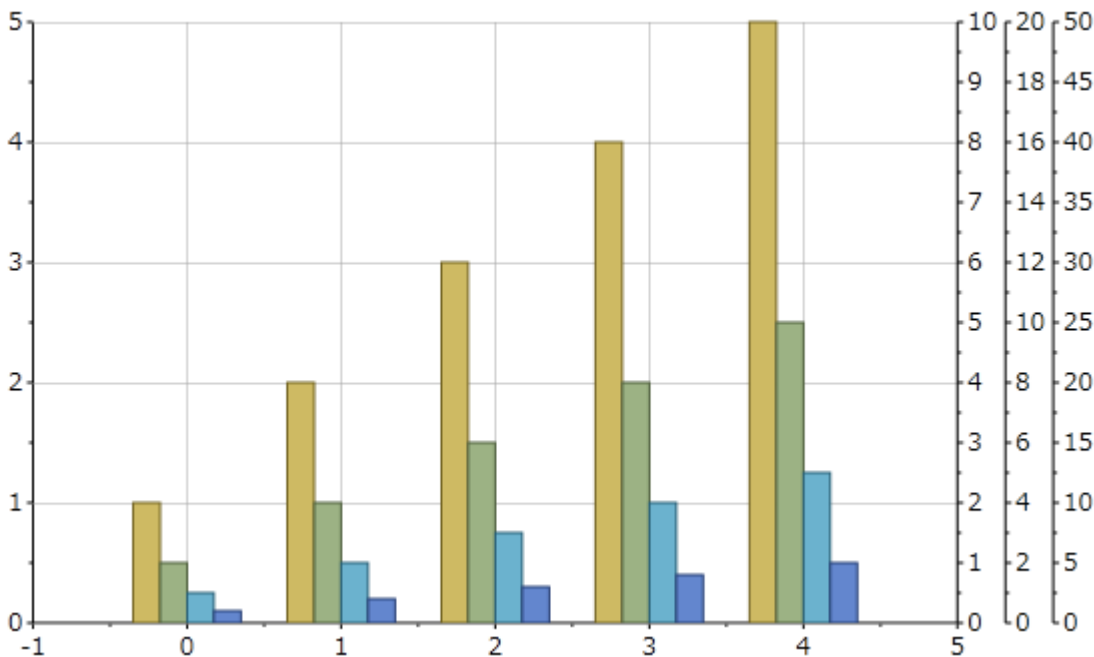
```
C#
clChart1.View.Inverted = true;
```

複数の軸

次のような場合は、複数の軸を使用するのが普通です。

- データ型の混在した複数のデータ系列があり、それぞれ大きく異なる目盛が必要な場合
- データの値がデータ系列間で広範囲に分散している場合

次のグラフでは、メートル法とそれ以外の体系の両方で長さや温度を効果的に表示するため、5本の軸を使用しています。



複数の軸をグラフに追加するには、新しい **Axis** オブジェクトを追加して、そのタイプ (X、Y、または Z) を **AxisType** プロパティに

Chart for WPF/Silverlight

指定します。

次の XAML コードは、複数の Y 軸をグラフに追加する方法を示しています。

XAML

```
<clchart:C1Chart Margin="0" Name="c1Chart1">
  <clchart:C1Chart.View>
    <clchart:ChartView>
      <!-- 補助の y 軸 -->
      <clchart:Axis Name="ay2" AxisType="Y" Position="Far" Min="0" Max="10" />
      <clchart:Axis Name="ay3" AxisType="Y" Position="Far" Min="0" Max="20" />
      <clchart:Axis Name="ay4" AxisType="Y" Position="Far" Min="0" Max="50" />
    </clchart:ChartView>
  </clchart:C1Chart.View>
  <clchart:C1Chart.Data>
    <clchart:ChartData>
      <clchart:DataSeries Values="1 2 3 4 5" />
      <clchart:DataSeries AxisY="ay2" Values="1 2 3 4 5" />
      <clchart:DataSeries AxisY="ay3" Values="1 2 3 4 5" />
      <clchart:DataSeries AxisY="ay4" Values="1 2 3 4 5" />
    </clchart:ChartData>
  </clchart:C1Chart.Data>
</clchart:C1Chart>
```

独立した軸を同時に拡大縮小する

独立した軸を同時に拡大縮小するには、次のように PropertyChanged イベントを使用して、両方の軸の scale プロパティと value プロパティをリンクする必要があります。

C#

```
// ay2 は補助 y 軸とします
((INotifyPropertyChanged) chart.View.AxisY).PropertyChanged += (s, e) =>
{
    if (e.PropertyName == "Scale")
    {
        ay2.Scale = chart.View.AxisY.Scale;
    }
    else if (e.PropertyName == "Value")
    {
        ay2.Value = chart.View.AxisY.Value;
    }
};
```

独立した軸を同時に拡大縮小する

独立した軸を同時に拡大縮小するには、次のように PropertyChanged イベントを使用して、両方の軸の scale プロパティと value プロパティをリンクする必要があります。

C#

```
// ay2 は補助 y 軸とします
((INotifyPropertyChanged) chart.View.AxisY).PropertyChanged += (s, e) =>
{
    if (e.PropertyName == "Scale")
```

```

{
    ay2.Scale = chart.View.AxisY.Scale;
}
else if (e.PropertyName == "Value")
{
    ay2.Value = chart.View.AxisY.Value;
}
};

```

軸の対数スケーリング

表示されるデータのばらつきが大きい場合、またはデータが1つのグラフ内で指数関数的に変化すると予想される場合は、少なくとも1つの軸に対数スケーリングを使用すると見やすくなることがあります。対数軸では、軸方向の同じ長さは同じ変化率を表します。一方または両方の軸に対数スケーリングが使用されたグラフは、対数スケールグラフと呼ばれます。

対数スケーリングを使用すると、値どうしの物理的な間隔は、値そのものではなく、値の対数が基準になります。これは、極めて広い範囲に渡る数量をグラフ化する場合や、幾何学的または指数関数的な関係を表す必要がある場合に役立ちます。

変化が直接的な単位で計測される**算術的グラフ**とは異なり、**対数スケールグラフ**では、変化が変化率によって表されます。たとえば、ドルの**対数スケールグラフ**では、1ドルから2ドルへの変化は 100 パーセントの変化なので、1ドルと2ドルの間隔は 50ドルと 100ドルの間隔と同じになります。**算術的グラフ**では、50ドルから 100ドルへの変化は 50ドルの変化であるのに対して、他方は1ドルなので、軸の上では 50ドルから 100ドルの方がはるかに大きな間隔になります。

よく使用される対数

対数は、整数と浮動小数点値を含む任意の底を使って表すことができます。最もよく使用される対数は、次の2種類です。

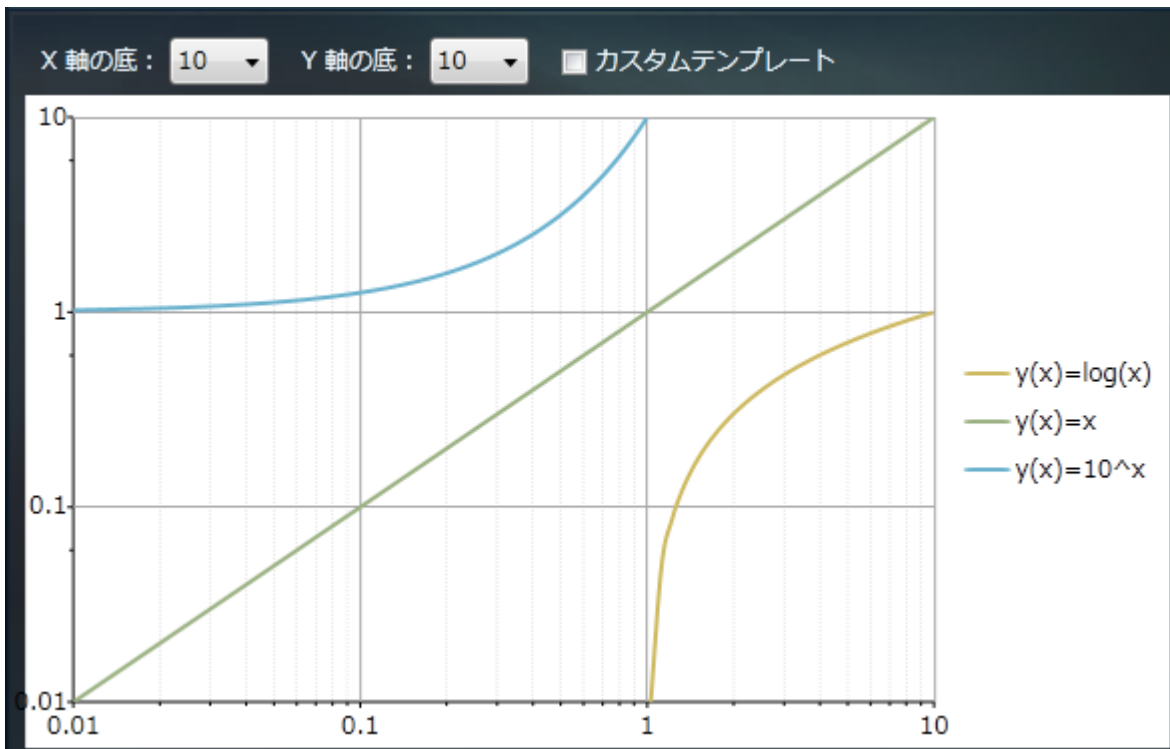
- **常用対数** – 底として 10 を使用します。したがって、 $\log 100 = 2$ です。
- **自然対数** – 底として数学上の定数 e を使用します。

対数の底

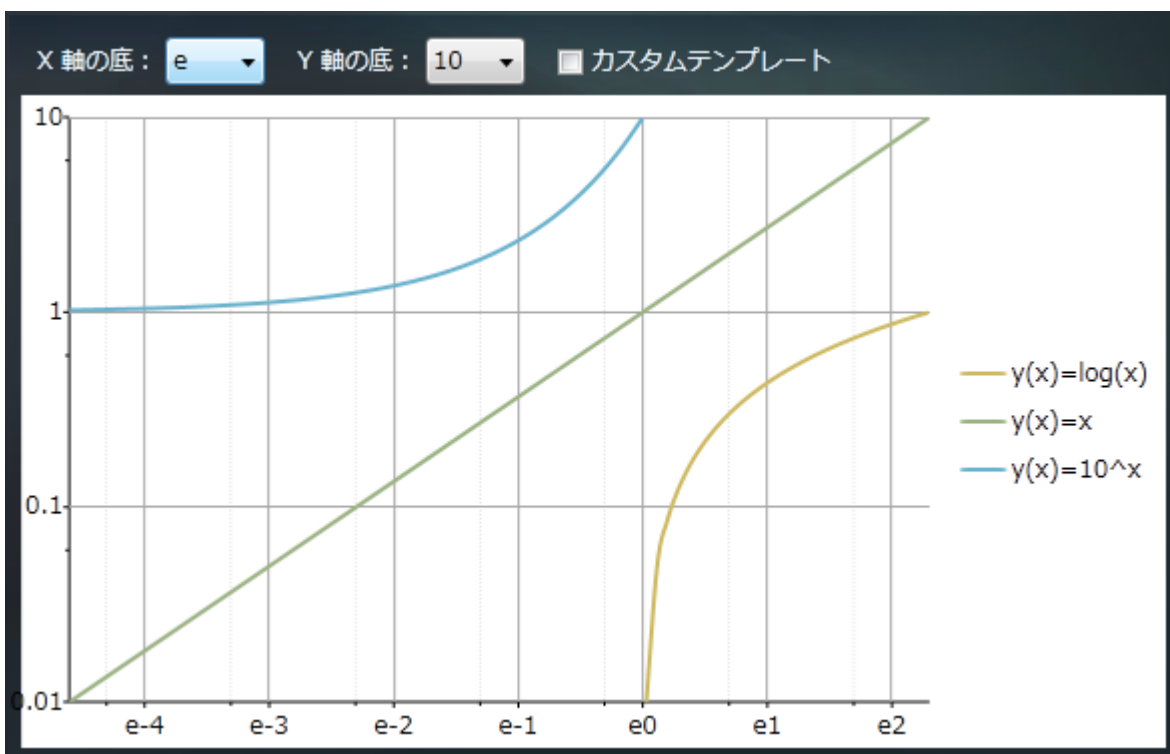
対数の底の値は、**LogBase** プロパティを使って指定できます。デフォルト値は、デフォルトの線形軸に対応する `double.NaN` です。自然対数は、底が e の対数です。底の値をゼロ以下にすると、対数スケーリングは数学的に意味を持たなくなります。

次の図に、X 軸と Y 軸の **LogBase** を常用対数の 10 に設定した場合の **C1Chart** を示します。

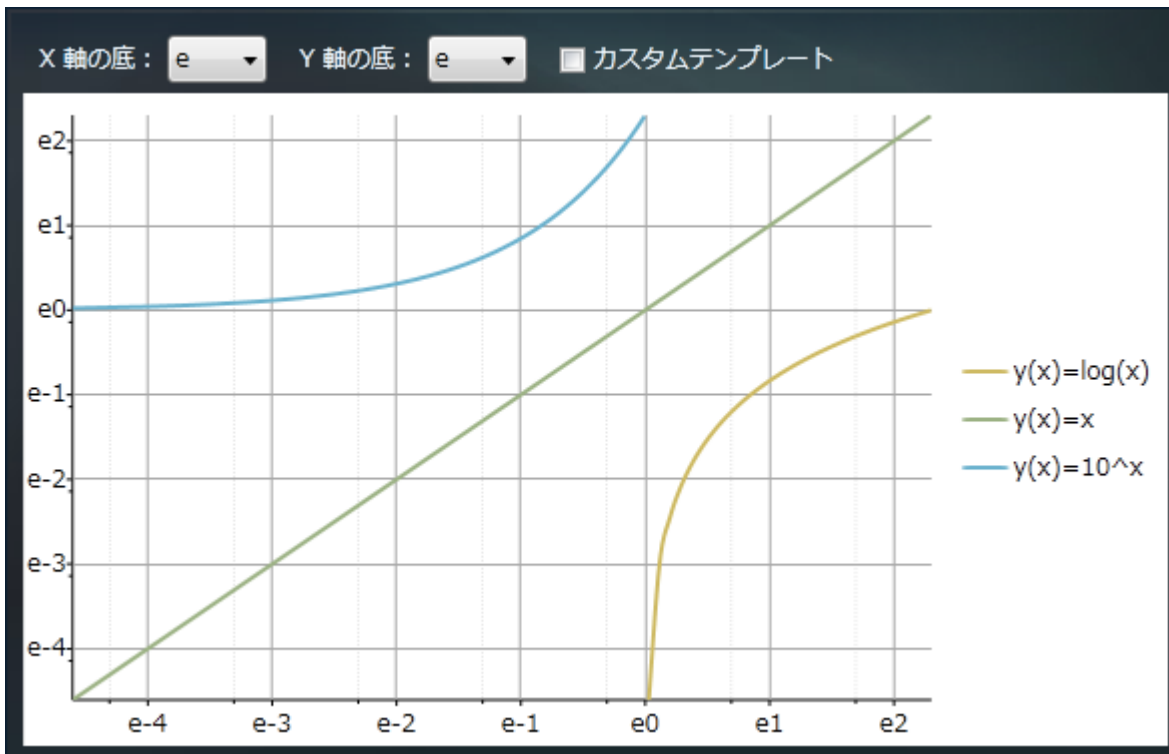
Chart for WPF/Silverlight



次の図に、X 軸の **LogBase** を e に設定し、Y 軸の **LogBase** を 10 に設定した場合の **C1Chart** を示します。



次の図に、X 軸の **LogBase** を e に設定し、Y 軸の **LogBase** を e に設定した場合の **C1Chart** を示します。



対数軸では、次のような基準にも従う必要があります。

- 対数軸は0より大きなデータ値のみを扱うため、0以下のデータはグラフ化されません(データ欠損として処理されます)。同じ理由で、軸やデータの最小値、最大値、原点の各プロパティを0以下に設定することはできません。
- 対数軸に対しては、軸目盛りの間隔、補助目盛りの間隔、および軸の精度を設定する各プロパティは無効です。
- X軸を対数にした場合、グラフタイプはプロット、バブル、面、HiLo、HiLoOpenClose、ローソク足のいずれかにする必要があります。Y軸を対数にした場合は、グラフタイプはプロット、バブル、面、ポーラ、HiLo、HiLoOpenClose、ローソク足、レーダー、塗りつぶしレーダーのいずれかにする必要があります。

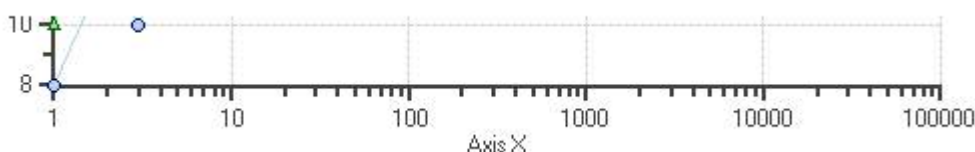
UnitMajor と対数軸

対数軸のスケールでは、**MajorUnit** が係数として各サイクルの底の値に掛けられて、対数の底の各サイクルにおける目盛りの間隔を決定します。つまり、(MajorUnit * 底サイクル値) は、各サイクル内の目盛りの増分とほぼ同じ値になります。対数の底が整数値の場合、結果は通常、そのままの値です。浮動小数点値の場合、目盛り値は線形スケールと同様に、適切な数値に丸められます。

UnitMajor と対数軸の詳細

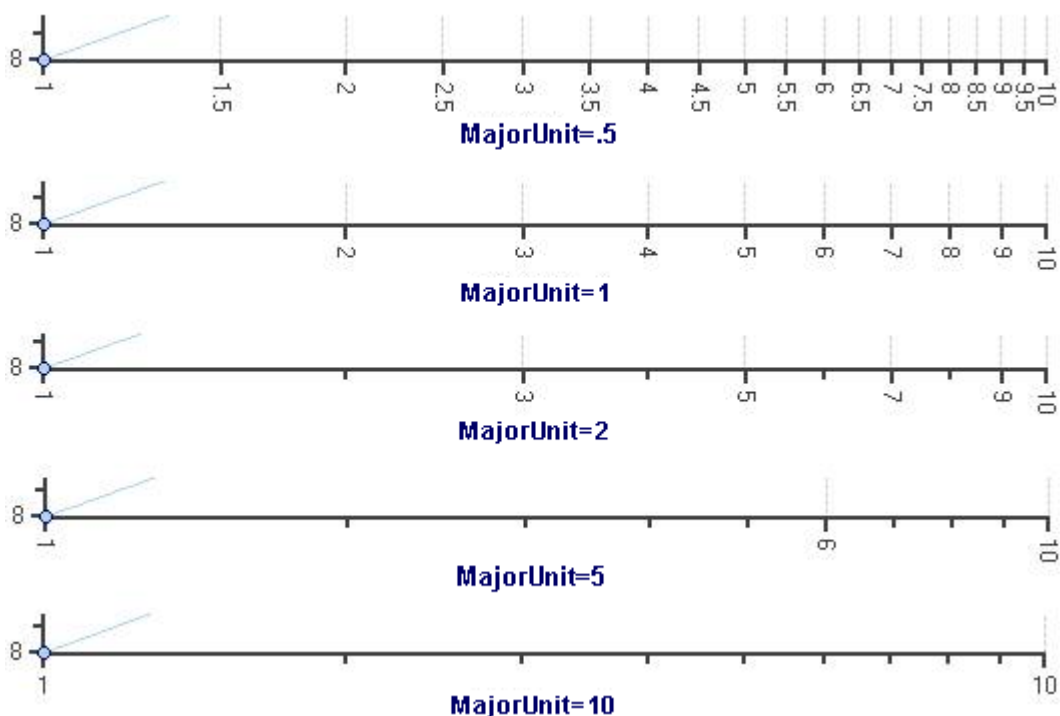
多くの場合、対数スケールを使用すると、グラフ軸の範囲は対数の底の複数のサイクルにまたがります。その場合、あるサイクルに適した値も、その前後のサイクルにとってはほとんど意味がないので、**MajorUnit** を通常のように線形的に指定しても意味がありません。**MajorUnit** の設定を有効に利用するには、対数の底の各サイクルに対応した値にする必要があります。

これで適切に対応できない場合は、この軸に使用できる単一値、固定値、または増分値を検討します。



上記の理由により、対数軸のグラフでは、**MajorUnit** には各サイクルの底の値分の1が指定されていると見なされます。次の例を考えます。

Chart for WPF/Silverlight



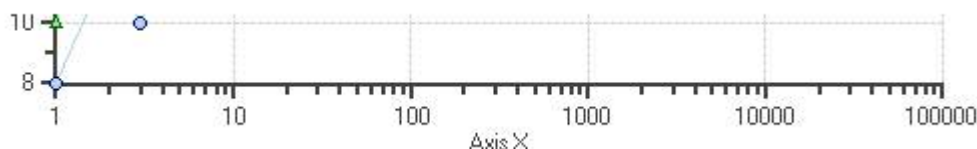
それぞれのケースの底サイクル値は1です。各サイクルにおいて、次の目盛り値 = 前の数値 + (サイクルの底の値 * MajorUnit) になります。**MajorUnit** の最大値は LogarithmicBase です。**MajorUnit** の自動設定値は、常に **LogBase** です。すべての目盛り値が計算されると、数値を読みやすくするために一定の丸めアルゴリズムが適用されます。この動作は多少複雑に見えますが、対数の底に任意の数を使用可能にしながら、目盛りを読みやすく保つようにした結果です。たとえば、上のプロットでは対数の底が 10 ですが、当然、底が 2、x などの対数も考えられます。

UnitMajor と対数軸

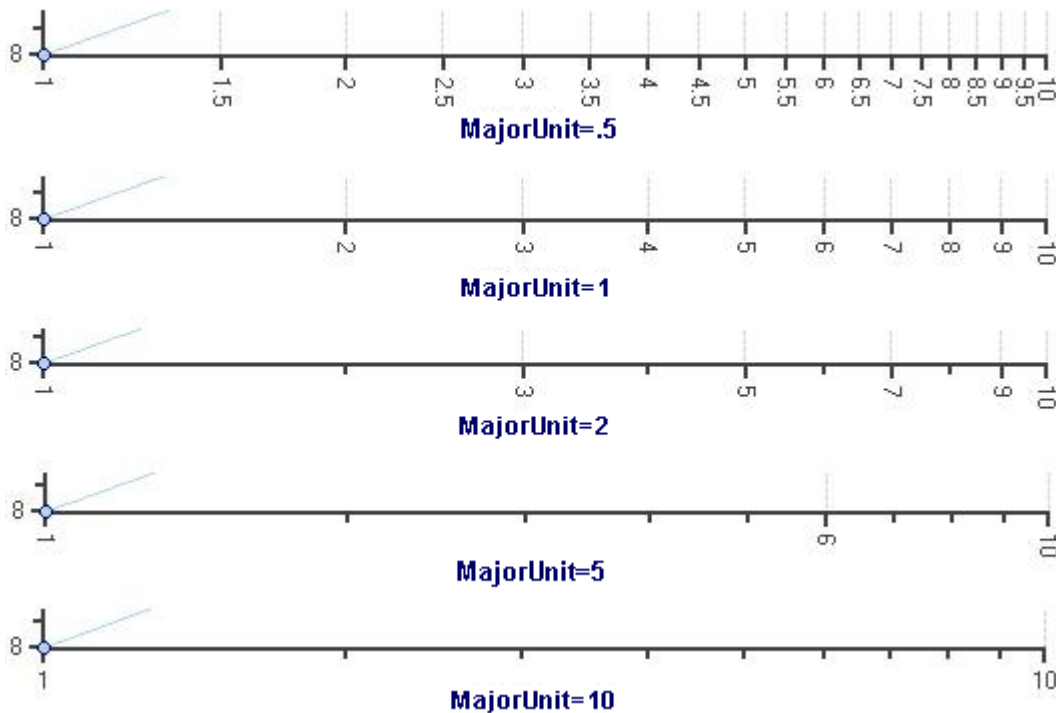
対数軸のスケージングでは、**MajorUnit** が係数として各サイクルの底の値に掛けられて、対数の底の各サイクルにおける目盛りの間隔を決定します。つまり、(MajorUnit * 底サイクル値) は、各サイクル内の目盛りの増分とほぼ同じ値になります。対数の底が整数値の場合、結果は通常、そのままの値です。浮動小数点値の場合、目盛り値は線形スケージングと同様に、適切な数値に丸められます。

UnitMajor と対数軸の詳細

多くの場合、対数スケールを使用すると、グラフ軸の範囲は対数の底の複数のサイクルにまたがります。その場合、あるサイクルに適した値も、その前後のサイクルにとってはほとんど意味がないので、**MajorUnit** を通常のように線形的に指定しても意味がありません。**MajorUnit** の設定を有効に利用するには、対数の底の各サイクルに対応した値にする必要があります。これで適切に対応できない場合は、この軸に使用できる単一値、固定値、または増分値を検討します。



上記の理由により、対数軸のグラフでは、**MajorUnit** には各サイクルの底の値分の 1 が指定されていると見なされます。次の例を考えます。



それぞれのケースの底サイクル値は1です。各サイクルにおいて、次の目盛り値 = 前の数値 + (サイクルの底の値 * MajorUnit) になります。**MajorUnit** の最大値は LogarithmicBase です。**MajorUnit** の自動設定値は、常に **LogBase** です。

すべての目盛り値が計算されると、数値を読みやすくするために一定の丸めアルゴリズムが適用されます。この動作は多少複雑に見えますが、対数の底に任意の数を使用可能にしながら、目盛りを読みやすく保つようにした結果です。

たとえば、上のプロットでは対数の底が 10 ですが、当然、底が 2、x などの対数も考えられます。

チャート凡例

チャート凡例を非表示にする

チャートの凡例をプログラムで非表示にするには、次の手順に従います。

XAML で凡例に名前を追加すると、`legend.Visibility = ...` というコードでその表示/非表示を変更できます。

XAML

```
<clchart:C1Chart x:Name="chart" >
  <clchart:C1ChartLegend x:Name="legend" />
  ...
</clchart:C1Chart>
```

凡例の方向と位置の変更

チャートの凡例をチャートコントロールの下中央の位置に水平に表示するには、次のコードを使用します。

C#

```
C1ChartLegend.Orientation = Horizontal
C1ChartLegend.Position = C1.WPF.C1Chart.LegendPosition.BottomCenter;
```

グラフ表示

ChartView オブジェクトは、データが含まれるグラフの領域を表します(タイトルと凡例を除きますが、軸は含まれます)。View プロパティは、以下の主なプロパティを持つ **ChartView** オブジェクトを返します。

| プロパティ | 説明 |
|----------------------------|--|
| Axes | 軸のコレクションを取得します。x、y、z の各軸を保存します。これらの軸は、グラフの範囲(最小値、最大値、単位、線形/対数目盛)と軸の線、グリッド線、目盛記号、および軸のラベルの外観を規定します。 |
| AxisSize (WPF のみ) | プロット領域全体に対する軸の領域の相対サイズを取得または設定します。 |
| AxisX, AxisY, AxisZ | これらのプロパティは、それぞれグラフの軸の外観をカスタマイズできる Axis オブジェクトを返します。 |
| Margin (WPF のみ) | グラフ領域とプロット領域の間隔を指定できる Margin オブジェクトを返します。軸のラベルは、このスペースに表示されます。 |
| PlotRect | 軸の内側の領域の外観を制御する Rect オブジェクトを返します。 |
| ChartView.Style | グラフ領域の色と枠線を設定するプロパティが含まれます。 |

ChartView クラスの次のプロパティは、3D グラフでのみ適用されます。

| プロパティ | 説明 |
|-----------------------------|--|
| Camera (WPF のみ) | 3D 専用のカメラを取得または設定します。 |
| Lights (WPF のみ) | 周辺光 (Ambient light) や指向性の光 (Directional light) など、3D グラフの状況で使用される光源を指定します。 |
| Perspective (WPF のみ) | 視点の転換の値を取得または設定します。 |
| Ratio (WPF のみ) | プロット領域内の軸の比率を取得または設定します。 |
| Transform (WPF のみ) | 変換、回転、およびスケール転換など、3D のすべての転換を指定できます。 |

プロットの背景の設定

プロットの背景を設定すると、チャートを簡単にカスタマイズしたりコントラストを付けることができます。

次の XAML マークアップを使用して、背景を設定できます。

XAML

```
<cl:C1Chart.View>
  <cl:ChartView PlotBackground="#FF343434">
  </cl:ChartView>
</cl:C1Chart.View>
```

PlotBackground|tag=PlotBackground_Property プロパティをコードで設定することもできます。チャートを名前指定し、次のコードを **InitializeComponent()** メソッドに追加するだけです。

C#

```
chart.View.PlotBackground = Brushes.BlueViolet;
```

データ連結

Chart for WPF/Silverlight の **C1Chart** コントロールは、**System.Collections.IEnumerable** インターフェイスを実装するすべてのオブジェクト (XmlDataProvider、DataSet、DataView など) に連結できます。

以下に、**C1Chart** コントロールにデータを提供するために使用されるさまざまなデータ連結方法について説明します。

値のコレクション

グラフにデータを提供するには、いくつかの方法があります。1つめの方法は、**ValuesSource** プロパティを使用して値のコレクションを連結します。

IEnumerable インターフェイスをサポートする数値のコレクションは、すべてデータ系列のデータソースとして使用できます。各データ系列クラスは、データ連結のための適切なプロパティを持ちます。たとえば、**DataSeries** クラスは、データ連結に **ValuesSource** プロパティを使用します。

値のコレクションを **DataSeries** に連結するには

たとえば、リソースに次の配列があるとします。

XAML

```
<!--ソースの連結 -->
<x:Array xmlns:sys="clr-namespace:System;assembly=mscorlib"
  x:Key="array" Type="sys:Double">
  <sys:Double>1</sys:Double>
  <sys:Double>4</sys:Double>
  <sys:Double>9</sys:Double>
  <sys:Double>16</sys:Double>
</x:Array>
```

この配列をデータ系列に渡すには、次のコードを使用します。

XAML

```
<!--ターゲットの連結 -->
<clchart:C1Chart Name="chart">
  <clchart:C1Chart.Data>
    <clchart:ChartData ItemsSource="{Binding Source={StaticResource array},
Path=Items}">
      <clchart:DataSeries ValuesSource="{Binding Source={StaticResource
array}, Path=Items}"/>
    </clchart:ChartData>
  </clchart:C1Chart.Data>
</clchart:C1Chart>
```

データ値を1つの属性として指定することもできます。値はスペースで区切る必要があります。たとえば、次のように指定します。

XAML

```
<clc:DataSeries Values="1 2 9 16"/>
```

Chart for WPF/Silverlight

上のマークアップは、**DataSeries** の **ValuesSource** プロパティを「1 2 9 16」の値で指定される **DataSeries** オブジェクトの **Items** プロパティに宣言的に連結します。

オブジェクトのコレクション

各オブジェクトに数値プロパティが含まれるオブジェクトのコレクションがある場合は、データ連結を使用する必要があります。このデータ連結処理には、少なくとも次の2つのグラフプロパティが関連します。

- **ItemsSource** プロパティ – オブジェクトのコレクションが割り当てられるソース。
- **ValueBinding** プロパティ – グラフのデータ系列に対する値の連結を取得または設定します。どのオブジェクトプロパティがデータ値を提供するかを指定します。

リソースに次の点の配列があるとします。

XAML

```
<x:Array x:Key="points" Type="Point">
  <Point>0,0</Point>
  <Point>10,0</Point>
  <Point>10,10</Point>
  <Point>0,10</Point>
  <Point>5,5</Point>
</x:Array>
```

次の XAML コードは、2つのデータ系列を持つグラフを表します。1つは点の x 座標に連結され、もう1つは y 座標に連結されます。

XAML

```
<clc:C1Chart Name="chart2">
  <clc:C1Chart.Data>
    <clc:ChartData
      ItemsSource="{Binding Source={StaticResource points}, Path=Items}">
      <clc:DataSeries ValueBinding="{Binding Path=X}"/>
      <clc:DataSeries ValueBinding="{Binding Path=Y}"/>
    </clc:ChartData>
  </clc:C1Chart.Data>
</clc:C1Chart>
```

次のサンプルは、点の XY 座標を同時に使用する系列を表します。**XYDataSeries** クラスのインスタンスが x 座標と y 座標に対応する2組のデータ値を処理していることに注意してください。

XAML

```
<clc:XYDataSeries
  XValueBinding="{Binding Path=X}"
  ValueBinding="{Binding Path=Y}"/>
```

Observable コレクション

WPF/Silverlight には、ObservableCollection という特別な汎用コレクションクラスがあります。このコレクションは、項目が追加または削除されたり、リスト全体が更新される場合に、更新に関する通知を提供します。このクラスのインスタンスをチャートのデータソースとして使用すると、チャートはコレクション内で行われた変更を自動的に反映します。

コードで、System.Collections.ObjectModel 名前空間をページ(および C1.WPF.C1Chart または C1.Silverlight.C1Chart)に追加します。これには ObservableCollection が含まれます。

WPF

```
C#
using System.Collections.ObjectModel;
using C1.WPF.C1Chart;
```

Silverlight

```
C#
using System.Collections.ObjectModel;
using C1.Silverlight.C1Chart;
```

次に、Point 型の ObservableCollection を宣言します。これがチャートのデータソースになります。

```
C#
ObservableCollection<Point> points = new ObservableCollection<Point>();
```

設定済みのすべてのチャートデータ(存在する場合)をクリアしてから、ポイントコレクションにダミーデータを挿入します。

```
C#
//チャートデータをクリアします
c1Chart1.Data.Children.Clear();
//ダミーデータを作成します
points.Add(new Point(0, 20));
points.Add(new Point(1, 22));
points.Add(new Point(2, 19));
points.Add(new Point(3, 24));
points.Add(new Point(4, 29));
points.Add(new Point(5, 7));
points.Add(new Point(6, 12));
points.Add(new Point(7, 15));
```

次に、このコレクションに連結された XYDataSeries を作成し、チャートに追加します。

```
C#
//C1Chart データ系列を設定します
XYDataSeries ds = new XYDataSeries();
ds.Label = "Series 1";
//データ系列をコレクションに連結します
ds.ItemsSource = points;
//ItemsSource を使用する場合は、連結を設定することが重要です
ds.ValueBinding = new Binding("Y");
ds.XValueBinding = new Binding("X");
//データ系列をチャートに追加します
c1Chart1.Data.Children.Add(ds);
```

Chart for WPF/Silverlight

ポイントのコレクションをデータ系列の `ItemsSource` に直接連結することができます。また、`Point` オブジェクトの `X` および `Y` フィールドに `ValueBinding (Y)` と `XValueBinding` を指定することも重要です。カスタムビジネスオブジェクトの場合と同様に、データ系列の値を目的のフィールドに連結する必要があります。その後、データ系列をチャートのデータコレクションに追加します。この方法によれば、複数のデータ系列を簡単に追加できます。

データコンテキストのバインディング

複数のプロパティを同じソースにバインドする場合は、**DataContext** プロパティを使用します。**DataContext** プロパティは、データの有効範囲を確立するための便利な方法を提供します。

C1Chart コントロールでは、項目のソースが設定されていないときに **DataContext** プロパティを **ItemsSource** として使用します。項目のソースとして使用するには、**DataContext** が **IEnumerable** である必要があります。

以下のトピックでは、**DataContext** を **double** および **Point** として使用する方法を説明します。

double の配列としてのデータコンテキスト

次のコードは、データコンテキストを **double** の配列として使用する方法を示しています。

```
C#
c1Chart1.Reset(true);
c1Chart1.DataContext = new double[] { 1, 2, 3, 4, 5 };
c1Chart1.ChartType = ChartType.Column;
```

Point の配列としてのデータコンテキスト

次のコードは、データコンテキストを **Point** の配列として使用する方法を示しています。

```
C#
c1Chart1.Reset(true);
c1Chart1.DataContext = new Point[] { new Point(1, 1), new Point(2, 2), new Point(3, 4), new Point(4, 1) };
c1Chart1.ChartType = ChartType.LineSymbols;
```

データ系列のバインディング

C1Chart では、グラフ上にプロットするプロパティの指定に使用される、次のような各種のバインディングが提供されています。

- **項目名のバインディング**: グラフのデータについて項目名のバインディングを指定します。
- **系列のバインディング**: データ系列の値バインディングのコレクション。自動生成時にデータ系列が作成されるコレクション内の各バインディングに適用します。
- **X 値のバインディング**: グラフのデータ系列について `x` 値のバインディングを指定します。

項目名のバインディング

ItemNameBinding プロパティが使用されるときに、グラフのデータについて項目名のバインディングを指定します。次の例は、ターゲットオブジェクトに対して **bindings** メソッドを呼び出します。

C#

```

ChartBindings bindings = new ChartBindings();
bindings.ItemNameBinding = new Binding("Name");
bindings.SeriesBindings.Add(new Binding("Input"));
bindings.SeriesBindings.Add(new Binding("Output"));
chart.Bindings = bindings;
chart.DataContext = new InOut[]
{
    new InOut() { Name = "n1", Input = 90, Output = 110},
    new InOut() { Name = "n2", Input = 80, Output = 70},
    new InOut() { Name = "n3", Input = 100, Output = 100},
};
// ここで、InOut は次のように定義されています。
public class InOut
{
    public string Name { get; set; }
    public double Input { get; set; }
    public double Output { get; set; }
}

```

X 値のバインディング

X 値のバインディングは、**XBinding** プロパティが使用されるときにグラフのデータ系列について x 値のバインディングを指定します。次の例は、**XBinding** プロパティを使用して、データ系列に x 値のバインディングを設定します。

C#

```

ChartBindings bindings = new ChartBindings();
bindings.XBinding = new Binding("X");
bindings.SeriesBindings.Add(new Binding("Y"));
chart.Bindings = bindings;
chart.DataContext = new Point[] { new Point(1, 0),
new Point(2, 2), new Point(3, 1), new Point(5, 3) };

```

DataSet の DataTable へのグラフのバインド

データテーブルからグラフを作成するサンプルコードを示します。

コードの場合

VisualBasic

```

Private _dataSet As DataSet
Private Sub Window_Loaded(ByVal sender As Object, ByVal e As RoutedEventArgs)
    ' コレクションを作成して、データセットに入力
    Dim mdbFile As String = "c:\db\nwind.mdb"
    Dim connString As String = String.Format("Provider=Microsoft.Jet.OLEDB.4.0; Data
Source={0}", mdbFile)
    Dim conn As New OleDbConnection(connString)
    Dim adapter As New OleDbDataAdapter("SELECT TOP 10 ProductName, UnitPrice FROM

```

Chart for WPF/Silverlight

```
Products " & vbCr & vbLf & " ORDER BY UnitPrice;", conn)

_dataSet = New DataSet()
adapter.Fill(_dataSet, "Products")

' データテーブルの行をグラフデータのソースとして設定
c1Chart1.Data.ItemsSource = _dataSet.Tables("Products").Rows
End Sub
```

C#

```
DataSet _dataSet;
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    // コレクションを作成して、データセットに入力
    string mdbFile = @"c:\db\nwind.mdb";
    string connString = string.Format(
        "Provider=Microsoft.Jet.OLEDB.4.0; Data Source={0}",
        mdbFile);
    OleDbConnection conn = new OleDbConnection(connString);
    OleDbDataAdapter adapter = new OleDbDataAdapter( "SELECT TOP 10 ProductName,
UnitPrice FROM Products ORDER BY UnitPrice;", conn);
    _dataSet = new DataSet();
    adapter.Fill(_dataSet, "Products");
    // データテーブルの行をグラフデータのソースとして設定
    c1Chart1.Data.ItemsSource = _dataSet.Tables["Products"].Rows;
}
```

XAML の場合

XAML

```
<clchart:C1Chart.Data>
  <clchart:ChartData ItemNameBinding="{Binding Path=[ProductName]}">
    <clchart:DataSeries ValueBinding="{Binding Path=[UnitPrice]}" />
  </clchart:ChartData>
</clchart:C1Chart.Data>
```

データポイントコンバータ

DataPointConverter クラスは、テンプレートの複雑なポイントラベルを xaml で作成するために便利です。DataPointConverter は、コンバータパラメータを使用して、現在のデータポイントのプロパティに基づく文字列を生成します。コンバータパラメータ文字列には、次のキーワードを挿入できます。これらのキーワードは、各ポイントの実際のプロパティ値に置き換えられます。

- #Values - データポイントの y 座標。
- #XValues - データポイントの x 座標 (XYDataSeries の場合)。
- #PointIndex - データポイントのインデックス。
- #SeriesIndex - データ系列のインデックス。
- #SeriesLabel - データ系列のラベル。
- #NewLine - 改行。

キーワードは、先頭に # を付け、中かっこで囲む必要があります。次の文字列書式の {#Values:0.0} のように、かっこの中にオプションの書式文字列を追加できます。

次の xaml は、DataPointConverter クラスの使用方法を示します。

XAML

```
<clchart:C1Chart Name="chart" ChartType="LineSymbols" Margin="20" >
  <clchart:C1Chart.Resources>
    <clchart:DataPointConverter x:Key="cnv"/>
  </clchart:C1Chart.Resources>
  <clchart:C1Chart.Data>
    <clchart:ChartData>
      <clchart:XYDataSeries Label="S1"
        XValues="1,2,3,4,5,6,7" Values="1,2,3,4,3,4,2" >
        <clchart:XYDataSeries.PointLabelTemplate>
          <DataTemplate>
            <Border BorderBrush="Black" BorderThickness="0.5"
              Background="#70FFFFFF"
              clchart:PlotElement.LabelAlignment="MiddleCenter">
              <TextBlock>
                <TextBlock.Text>
                  <Binding Converter="{StaticResource cnv}">
                    <Binding.ConverterParameter>
                      {#SeriesLabel}{#NewLine}
                      X={#XValues:0.0},Y={#Values:0.0}{#NewLine}
                      SI={#SeriesIndex},PI={#PointIndex}
                    </Binding.ConverterParameter>
                  </Binding>
                </TextBlock.Text>
              </TextBlock>
            </Border>
          </DataTemplate>
        </clchart:XYDataSeries.PointLabelTemplate>
      </clchart:XYDataSeries>
    </clchart:ChartData>
  </clchart:C1Chart.Data>
</clchart:C1Chart>
```

データラベル

データラベルは、チャート内のデータポイントに関連付けられたラベルです。グラフにデータラベルを使用すると、特定のポイントが属する系列や特定のポイントの正確な値がわかりやすくなるため、便利な場合があります。

C1Chart はデータラベルをサポートします。各データ系列には **PointLabelTemplate** プロパティがあります。このプロパティは、各ポイントの隣に表示されるビジュアル要素を示します。通常、**PointLabelTemplate** は XAML リソースとして定義され、XAML またはコードからグラフに割り当てることができます。

DataTemplate を追加すると、データをどのように表示するかという視覚属性と、表示されたデータにデータ連結からどのようにアクセスするかという両方を決めることができます。

PointLabelTemplate を XAML リソースとして定義するには、リソースディクショナリを作成し、そのリソースディクショナリに DataTemplate リソースを追加します。これにより、Window.xaml ファイルから、DataTemplate リソースにアクセスできます。

新しいリソースディクショナリを追加するには、次の手順に従います。

Chart for WPF/Silverlight

1. ソリューションエクスプローラで、プロジェクトを右クリックし、[追加]をポイントして、[リソースディクショナリ]を選択します。[新しい項目の追加]ダイアログボックスが表示されます。
2. [名前]テキストボックスに、辞書の名前 Resources.xaml を入力し、[追加]ボタンをクリックします。Resources.xaml がプロジェクトに追加され、コードエディタで開かれます。

ラベルを作成するには、ラベルテンプレートを作成し、そのテンプレートに **PointLabelTemplate** を割り当てる必要があります。

各データポイントのプロットのレンダリング時に、指定したテンプレートに基づいてラベルが作成されます。ラベルの **DataContext** プロパティは、ポイントに関する情報を提供する現在の **DataPoint** インスタンスに設定します。データ連結を使用している場合は、より簡単にこの情報をラベルに表示できます。

次は、ポイントの値を表示するラベルテンプレートのサンプルです。

XAML

```
<DataTemplate x:Key="lbl">
    <TextBlock Text="{Binding Path=Value}" />
</DataTemplate>
```

リソースを定義したら、そのリソースをプロパティ値として参照できます。それには、キー名を示すリソースマークアップ拡張構文を使用します。

テンプレートをデータ系列に割り当てるには、次のように **PointLabelTemplate** プロパティを設定します。

XAML

```
<clchart:DataSeries PointLabelTemplate="{StaticResource lbl}" />
```

これは標準のデータテンプレートなので、複雑なラベルを作成できます。たとえば、次のサンプルテンプレートは、データポイントの **XY** 座標を表示する **XY** グラフのデータラベルを定義します。

このテンプレートでは、2行2列の標準グリッドをコンテナとして使用します。ポイントの **x** 値は、**DataPoint** クラスのインデクサを使用して取得されます。インデクサを使用して、複数のデータセットをサポートする **XYDataSeries** クラスなどのデータ系列クラスの値を取得できます。

XAML

```
<DataTemplate x:Key="lbl">
    <!-- Grid 2x2 with black border -->
    <Border BorderBrush="Black">
        <Grid>
            <Grid.ColumnDefinitions>
                <ColumnDefinition />
                <ColumnDefinition />
            </Grid.ColumnDefinitions>
            <Grid.RowDefinitions>
                <RowDefinition />
                <RowDefinition />
            </Grid.RowDefinitions>
            <!-- x-coordinate -->
            <TextBlock Text="X=" />
            <TextBlock Grid.Column="1" Text="{Binding Path=[XValues]}" />
            <!-- y-coordinate -->
            <TextBlock Grid.Row="1" Text="Y=" />
            <TextBlock Grid.Row="1" Grid.Column="1" Text="{Binding Path=Value}"
        />
    </Grid>
```

```
</Border>
</DataTemplate>
```

数値データを表示する場合は、出力値を書式設定することがよくあります。静的クラス **Format** を使用して、XAML コード内で標準の .Net 書式文字列を指定できます。たとえば、次のサンプルコードは、パーセント値を書式設定するためにコンバータを使用しています。

XAML

```
<DataTemplate x:Key="lbl1">
  <TextBlock Text="{Binding Path=PercentageSeries,
    Converter={x:Static clchart:Converters.Format},
    ConverterParameter=#.##%}" />
</DataTemplate>
```

グラフのデータ系列

C1Chart で特に重要なオブジェクトの1つは、データ系列です。データ系列には、グラフに含まれるデータや多数の重要なデータ関連プロパティがすべて含まれます。

さまざまなデータ系列クラス

様々なデータ型を効果的に処理するには、C1Chartでは次のdataserieクラスが提供されています。

- **BubbleSeries**
- **DataSeries**
- **HighLowOpenCloseSeries**
- **HighLowSeries**
- **XYDataSeries**
- **XYZDataSeries**

DataSeries クラス内の**Label**プロパティは、グラフの凡例に使用される系列名を表します。

同じ基本クラス **DataSeries** クラスからいくつかの **DataSeries** クラスが継承されています。各クラスは、それぞれのデータ特性に基づいて、いくつかのデータセットを組み合わせています。たとえば、**XYDataSeries** は、x および y 座標に対応する2つのデータセット値を提供します。次の表に、使用可能なデータ系列クラスを一覧します。

表 2 データ系列クラス

| クラス | Values プロパティ | ValueBinding プロパティ |
|----------------------|--|--|
| DataSeries | Values 、 ValuesSource | ValueBinding |
| XYDataSeries | Values 、 ValuesSource XValues 、 XValuesSource | ValueBinding XValueBinding |
| HighLowSeries | Values 、 ValuesSource XValues 、 XValuesSource HighValues 、 HighLowSeries.HighValuesSource LowValues 、 HighLowSeries.LowValuesSource | ValueBinding XValueBinding HighValueBinding LowValueBinding |

| | | |
|-------------------------------|--|---|
| HighLowOpenCloseSeries | Values、ValuesSource XValues、XValuesSource HighValues、 HighLowSeries.HighValuesSource LowValues、 HighLowSeries.LowValuesSource OpenValues、 HighLowOpenCloseSeries.OpenValuesSource CloseValues、 HighLowOpenCloseSeries.CloseValuesSource | ValueBinding XValueBinding HighValueBinding LowValueBinding OpenValueBinding CloseValueBinding |
|-------------------------------|--|---|

各データ系列クラスは、プロットのために独自のデフォルトテンプレートを持つ場合があります。たとえば、**HighLowOpenCloseSeries** は、高値、安値、始値、終値を示す1組の線として金融関連データを表示します。

グラフのデータ系列の外観

各データ系列の外観は、**DataSeries** クラスの3つのプロパティグループ、**Symbol**、**Connection**、および **ConnectionArea** によって決まります。これらのプロパティは、グラフタイプに応じてグラフの異なった部分に影響を与えます。

Symbol プロパティは、各データポイントに表示される記号の形状、サイズ、輪郭、および塗りつぶしの各プロパティを決定します。それらは、**折れ線グラフ**、**エリアグラフ**、**XY プロットグラフ**など、記号を表示するグラフタイプに適用されます。**Symbol** プロパティでは、**横棒グラフ**と**縦棒グラフ**の棒の外観も制御します。

Connection プロパティは、2つのデータポイント間に引かれる線の輪郭と塗りつぶしのプロパティを決定します。それらは、データ系列のポイント全体に適用されます。折れ線グラフの場合、**Connection** は各データポイントを結ぶ線であり、エリアグラフの場合は、データポイントの下の輪郭を含む領域です。

DataSeries と XYDataSeries の相違点

DataSeries には、データ値の論理セットが1つだけ含まれます (Values[y の値])。

この場合、x の値は自動的に生成され(0、1、2、...)、**Data.ItemNames** プロパティを使用して x 軸のラベルを指定することもできます。

XYDataSeries には、データ値のセットが2つ含まれます (Values[y の値]と XValues)。

折れ線グラフまたは円グラフにギャップを表示する

デフォルトでは、データ値に欠損(double.NaN)がある場合、チャートは単にその値をスキップし、次の有効なデータポイントまで線を引きます。

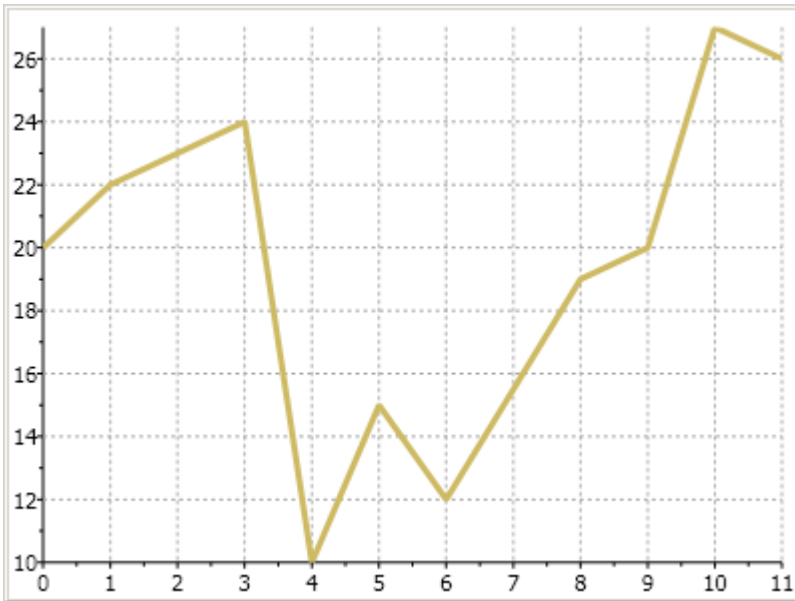
この動作を変更し、欠損値の場所にギャップを入れるには、**Display = ShowNaNGap** を設定します。

たとえば、次の XAML コードでは、**DataSeries** 内で欠損が指定されています。

XAML

```
<clchart:C1Chart Name="c1Chart1" ChartType="Line">
  <clchart:C1Chart.Data>
    <clchart:ChartData>
      <clchart:DataSeries Values="20 22 NaN 24 15 NaN 27 26"
        ConnectionStrokeThickness="3" />
    </clchart:ChartData>
  </clchart:C1Chart.Data>
</clchart:C1Chart>
```

Display プロパティが設定されていない場合、チャートは次のように表示されます。



折れ線グラフ上のグラフ線の上にギャップを入れるには、次のように **Display** プロパティを ShowNaNGap に設定できます。

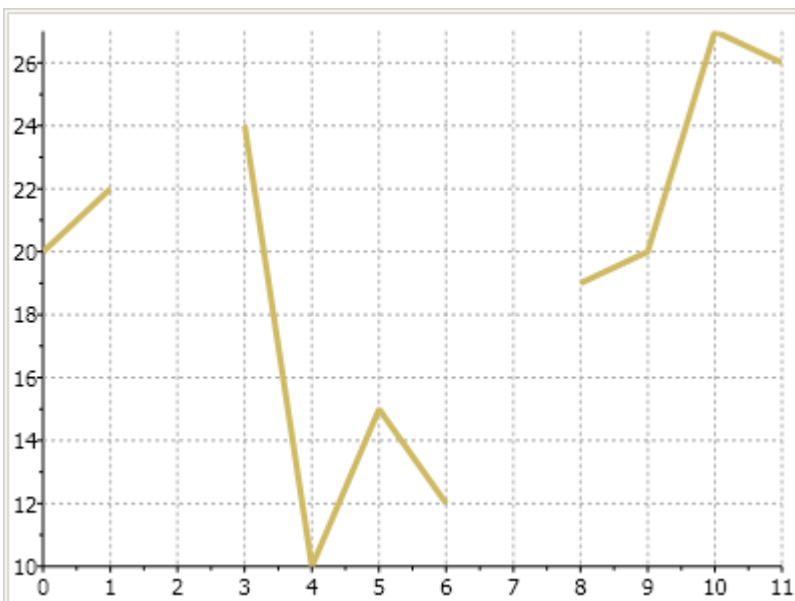
Visual Basic

```
Me.C1Chart1.Data.Children(1).Display = C1.WPF.C1Chart.SeriesDisplay.ShowNaNGap
```

C#

```
this.C1Chart1.Data.Children[1].Display = C1.WPF.C1Chart.SeriesDisplay.ShowNaNGap;
```

次のように、折れ線グラフのグラフ線の上にギャップが入ります。



グループ化と集計

Chart for WPF/Silverlight

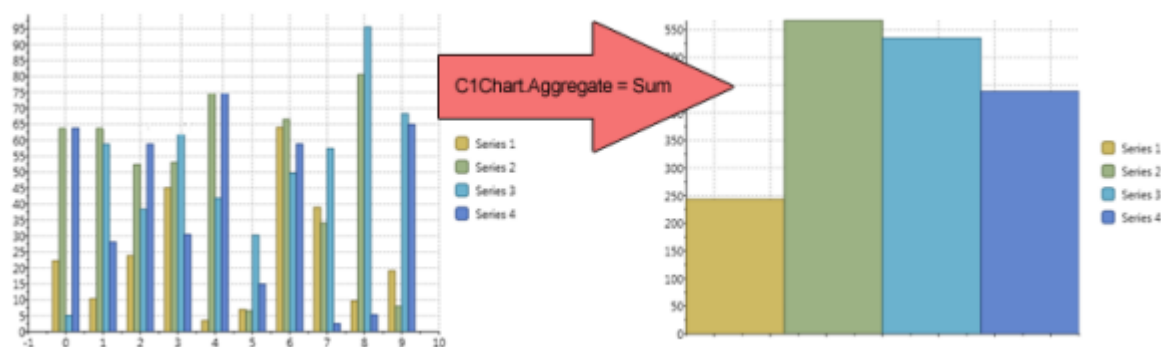
C1Chart コントロールは、組み込みのグループ化と集計をサポートします。これにより、データを自分でグループ化するという余分な作業を行わなくても、チャートに要約データを表示することができます。

DataSeries の集計

C1Chart コントロールでは、1つのプロパティを設定するだけで、自動的に各データ系列をグループ化して集計し、それを1つの値としてプロットできます。これは、サポートされている集計関数(合計、平均、最小、最大、分散、標準偏差など)のいずれかを使用して、1つの系列のすべての値を結合して1つの値を作成します。

| メンバ名 | 説明 |
|----------------------|------------------------|
| None | 未加工の値(集計なし)。 |
| Sum | 各ポイントのすべての値の合計を計算します。 |
| Count | 各ポイントの値の数。 |
| Average | 各ポイントのすべての値の平均を算出します。 |
| Minimum | 各ポイントの最小値を取得します。 |
| Maximum | 各ポイントの最大値を取得します。 |
| Variance | 各ポイントの(標本)分散を取得します。 |
| VariancePop | 各ポイントの(母分散)分散を取得します。 |
| StandardDeviation | 各ポイントの(標本)標準偏差を取得します。 |
| StandardDeviationPop | 各ポイントの(母分散)標準偏差を取得します。 |

C1Chart コントロールで Aggregate プロパティを設定するだけで、すべての系列にグループ化を適用できます。たとえば、4つのデータ系列を含むチャートがあり、Aggregate プロパティを Sum に設定すると、結果は次のようになります。



データ系列の集計に使用されるマークアップは次のようになります。

XAML

```
<clchart:C1Chart x:Name="chart0" Height="350" Width="450" ChartType="Column"
Palette="Solstice" Foreground="Black" >
<!-- グラフに3つの系列を挿入します -->
    <clchart:C1Chart.Data>
        <clchart:ChartData>
            <clchart:ChartData.Children>
                <clchart:DataSeries Label="Revenue" Aggregate="Sum"
Values="1200, 1205, 400, 1410" ></clchart:DataSeries>
                <clchart:DataSeries Label="Expense" Aggregate="Sum"
```



```

Values="400, 300, 300, 210">< /clchart:DataSeries>
        <clchart:DataSeries Label="Profit" Aggregate="Sum"
Values="790, 990, 175, 1205" ></clchart:DataSeries>
        </clchart:ChartData.Children>
    </clchart:ChartData>
</clchart:C1Chart.Data>
</clchart >

```

DateTime のグループ化

C1Chart コントロールでは、任意のデータ系列に対してカスタム集計を行うことができます。**AggregateGroupSelector** プロパティに独自のカスタム関数を定義することで、**C1Chart** コントロールで自在にデータをグループ化できます。たとえば、ある日付フィールドに基づくグループ化を行い、月単位や年単位で値を集計できます。独自の値の範囲やカテゴリを設定して、データポイントをグループ化することもできます。

このトピックは、既に XAML で "C1Chart1" という名前の C1Chart コントロールを作成していることを前提としています。XAML でコントロールを作成する方法については、「クイックスタート」または「概念と主要なプロパティ」を参照してください。

カスタム集計関数を作成するには、まず、集計するデータを作成する必要があります。Value(double)と Date(DateTime)の2つのプロパティを含む簡単なビジネスオブジェクトを作成します。次の例では、このビジネスオブジェクトに SampleItem と名前を付けます。参考のため、このオブジェクトをこのトピックの最後に記載します。

ランダムなデータから成る ObservableCollection を作成します。

```

C#
Random rnd = new Random();
ObservableCollection<SampleItem> _items = new ObservableCollection<SampleItem>();
for(int i = 0; i < 400; i++)
{
    _items.Add(new SampleItem { Value = rnd.Next(0, 100), Date =
DateTime.Now.AddDays(i) });
}

```

次に、XYDataSeries を項目のコレクションに連結します。

```

C#
//データ系列を構成します
var ds = new XYDataSeries()
{
    ItemsSource = _items,
    ValueBinding = new Binding { Path = new PropertyPath("Value") },
    XValueBinding = new Binding { Path = new PropertyPath("Date") },
    Aggregate = Aggregate.Sum,
    AggregateGroupSelector = GroupSelectorByDate,
    Label = "Sales"
};

```

上のコードでは、Aggregate と AggregateGroupSelector の2つのキープロパティを設定しました。Aggregate プロパティは、チャートデータの集計に使用する関数を指定します。AggregateGroupSelector プロパティは、データ系列にグループ化選択キーを提供する関数を指定します。ただし、カスタム関数を設定する前に、チャートに系列を追加し、X 軸に日付を表示するように構成します。また、日付が正しく表示されるように、X 軸を IsTime に設定します。

```

C#

```

Chart for WPF/Silverlight

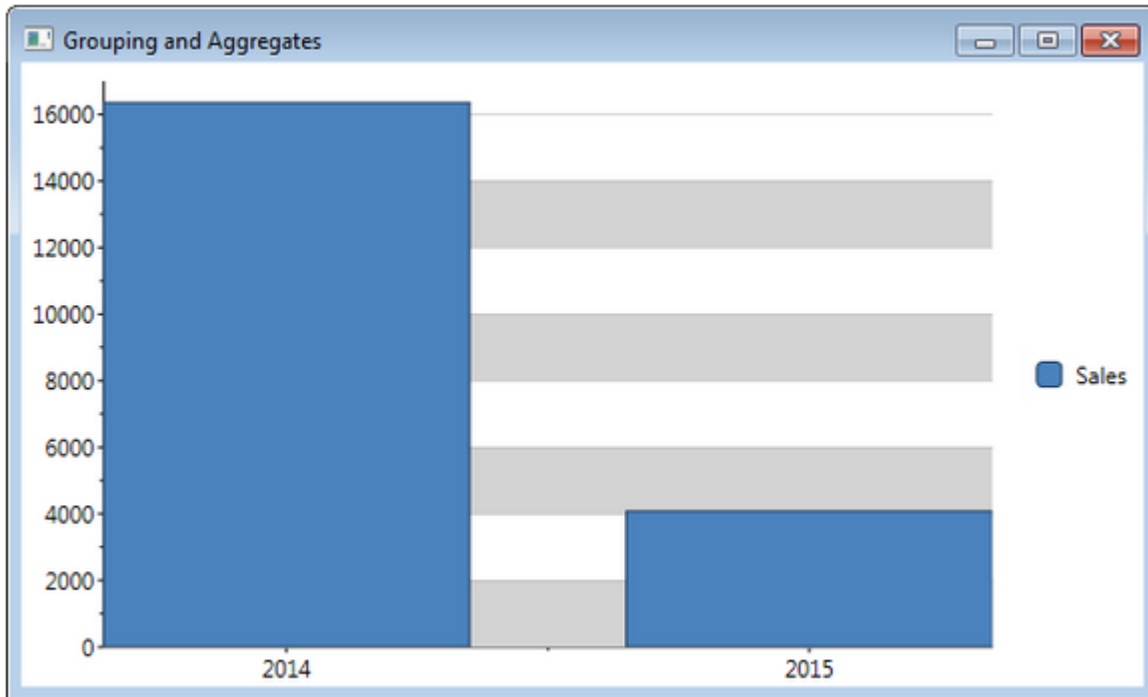
```
//チャートを構成します
c1Chart1.BeginUpdate();
c1Chart1.ChartType = ChartType.Column;
//データ系列を追加します
c1Chart1.Data.Children.Add(ds);
//書式を設定した時間軸を使用します
c1Chart1.View.AxisX.IsTime = true;
c1Chart1.View.AxisX.AnnoFormat = "yyyy";
c1Chart1.View.AxisX.UseExactLimits = true;
//スタイルを適用します
c1Chart1.View.AxisX.MajorGridStrokeThickness = 0;
c1Chart1.View.AxisY.MajorGridFill = new SolidColorBrush(Colors.LightGray);
c1Chart1.EndUpdate();
```

X 軸に日付を表示するには、IsTime プロパティを True に設定する必要があります。年単位でグループ化するため、年を省略しないで表示するように AnnoFormat プロパティを設定しました。

次のコードは、GroupSelectorByDate 関数を定義します。チャートの各データポイントに対してこの関数が呼び出され、そのデータが属するグループが決定されます。

```
C#
double GroupSelectorByDate(double x, double y, object o)
{
    //年を double として返します
    DateTime dt = x.FromOADate();
    //年単位でグループ化するために、日付の年を返します
    //また、AnnoFormat を "yyyy" に設定します
    return new DateTime(dt.Year, 1, 1).ToOADate();
}
```

このグループ選択関数は常に3つのパラメータを受け取り、常に double を返します。同じグループに属するデータポイントは、この関数から同じ値を返します。年単位でグループ化しているため、この関数は、年の最初の日に設定された新しい DateTime 値を返します。2014 年に発生する各データポイントは、この関数から同じ日付値(double)を返すため、同じグループに入れられます。



また、月単位でグループ化する場合は、コードを2行変更するだけです。

C#

```
double GroupSelectorByDate(double x, double y, object o)
{
    //年を double として返します
    DateTime dt = x.FromOADate();
    //月単位でグループ化するために、日付の年と月を返します
    return new DateTime(dt.Year, dt.Month, 1).ToOADate();
}
```

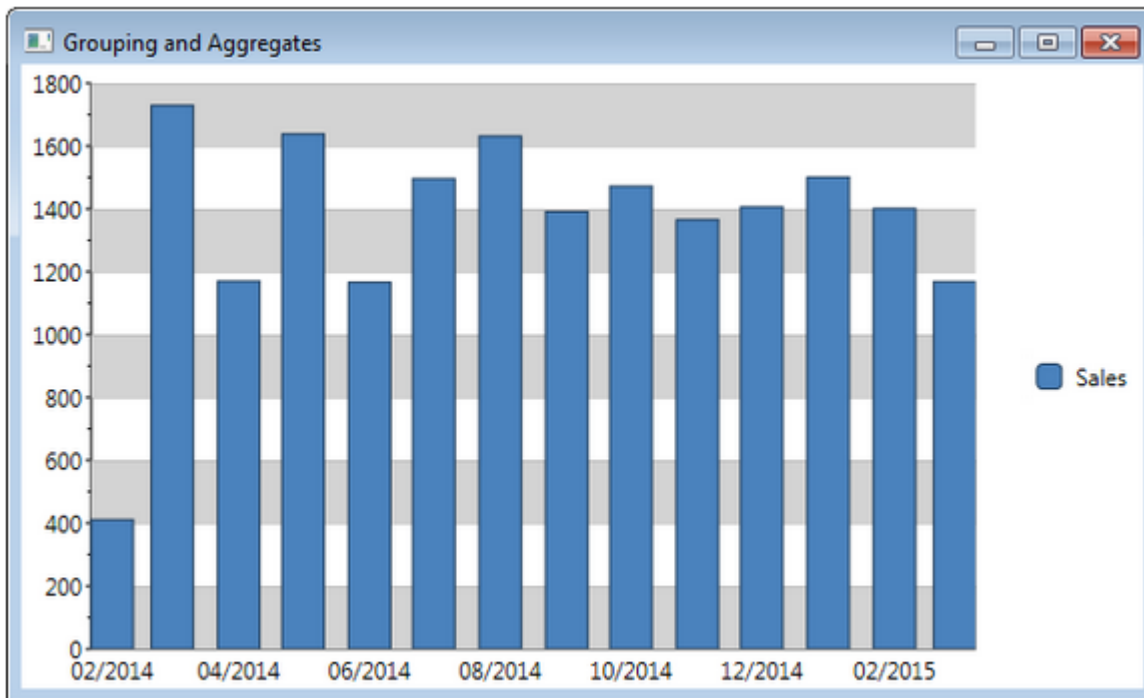
さらに、月が正しく表示されるように、AnnoFormat を変更する必要があります。

C#

```
c1Chart1.View.AxisX.AnnoFormat = "MM/yyyy";
```

生成されるチャートは次の図のようになります。

Chart for WPF/Silverlight



参考のために、次に SampleItem クラスを示します。

```
C#  
publicclass SampleItem  
{  
    publicdouble Value { get; set; }  
    public DateTime Date { get; set; }  
}
```

カスタムグループ化

Chart for WPF では、AggregateGroupSelector プロパティを使用して、カスタムグループ化関数とカスタム集計関数を作成できます。次の例では、カテゴリに基づくカスタム集計関数を作成する方法について説明します。WPF ストアアプリケーションの MainWindow.xaml ページから始めます。

まず、アプリケーションに C1Chart コントロールを追加し、"chart" と名前を付します。

```
XAML  
<c1chart:C1Chart Name="chart"></c1chart:C1Chart>
```

次に、汎用のボタンコントロールを追加し、Click イベントを設定します。

```
XAML  
<Button Content="New Data" Width="100" Click="Button_Click" />
```

コードビューに切り替えます。次の using 文をページの先頭に追加します。

WPF

```
using Cl.WPF.ClChart;
```

Silverlight

```
using Cl.Silverlight.ClChart;
```

次に、**MainWindow()** のコンストラクタを次のように編集します。

```
C#
public MainWindow()
{
    InitializeComponent();
    CreateSampleChart();
}
```

CreateSampleChart() メソッドを追加します。

```
C#
void CreateSampleChart()
{
}
}
```

CreateSampleChart() メソッド内で、項目名を格納する List オブジェクトを作成します。

```
C#
var keys = new List<string>{ "oranges", "apples", "lemons", "grapes" };
```

次に、連結 **DataSeries** を追加します。

```
C#
for (int i = 0; i < 2; i++)
{
    var ds = new DataSeries()
    {
        ItemsSource = SampleItem.CreateSampleData(40),
        ValueBinding = new Binding() { Path = new PropertyPath("Value") },
        Aggregate = Aggregate.Sum,
        Label = "s" + i
    };
}
```

AggregateGroupSelector 関数と、チャートに **DataSeries** を追加するコードを追加します。ここで、**AggregateGroupSelector** 関数は、この後に追加する **SampleItem** クラスから項目名を返します。

```
C#
ds.AggregateGroupSelector = (x, y, o) =>
{
    //カテゴリリストのインデックス
    return keys.IndexOf(((SampleItem)o).Name);
}
```

Chart for WPF/Silverlight

```
};  
chart.Data.Children.Add(ds);  
}  
chart.Data.ItemNames = keys;  
}
```

ユーザーがボタンをクリックするたびに、**Button_Click** イベントは、古いデータをクリアしてから新しいランダムデータを呼び出します。

C#

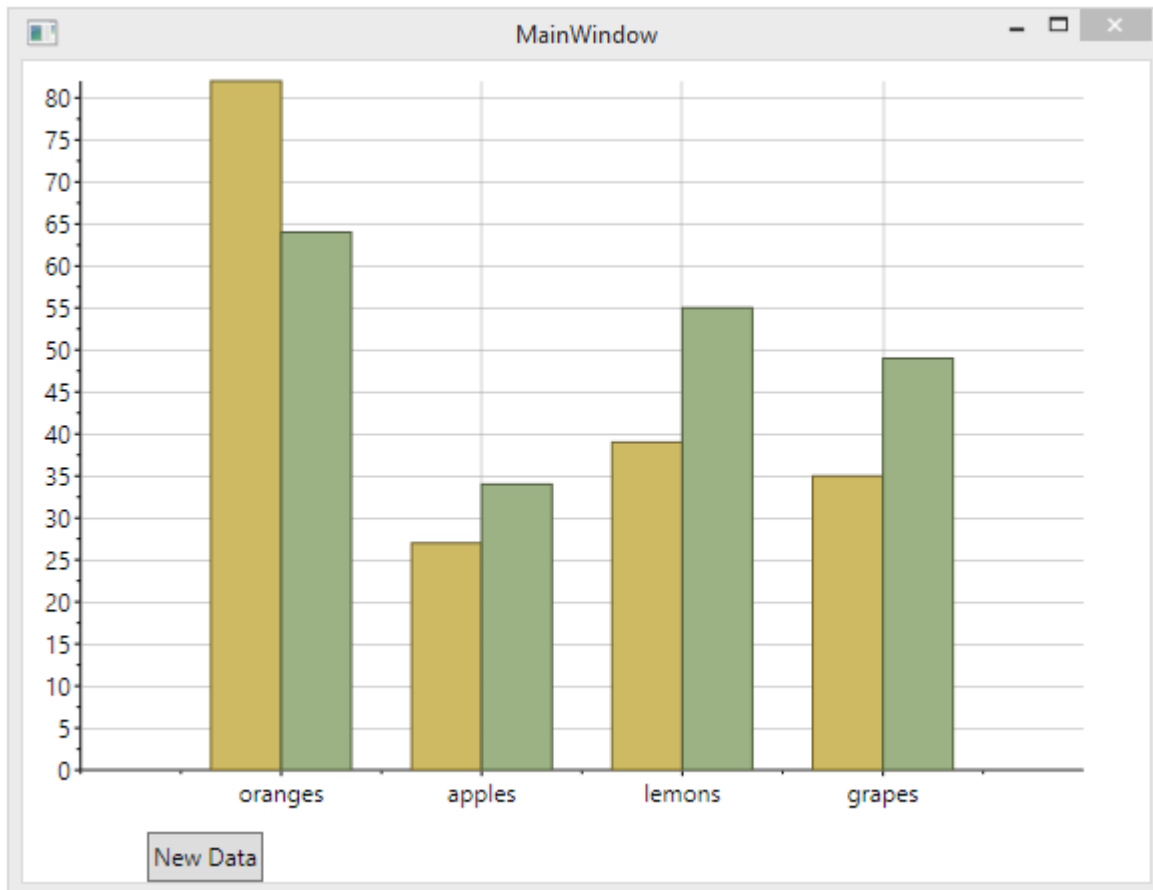
```
private void Button_Click(object sender, RoutedEventArgs e)  
{  
    chart.Data.Children.Clear();  
    CreateSampleChart();  
}
```

最後に、**SampleItem** クラスを追加します。これは、チャートコントロールにランダムデータを作成します。

C#

```
public class SampleItem  
{  
    public string Name { get; set; }  
    public double Value { get; set; }  
    static Random rnd = new Random();  
    public static SampleItem[] CreateSampleData(int cnt)  
    {  
        var names = new string[] { "oranges", "apples", "lemons", "grapes" };  
        var array = new SampleItem[cnt];  
        for (int i = 0; i < cnt; i++)  
        {  
            array[i] = new SampleItem() { Value = rnd.Next(1, 10), Name =  
names[rnd.Next(names.Length)] };  
        }  
        return array;  
    }  
}
```

上のコードとマークアップにより、次の画像のようなアプリケーションが表示されます。



エンドユーザー操作

C1Chart には、エンドユーザー向けのインタラクティブ動作を簡単に実装できるツールが組み込まれています。エンドユーザーは、マウスとシフトキーを組み合わせて使用して、グラフを詳しく調べたり、回転させたり、拡大／縮小したりできます。

インタラクティブ機能の管理の中心は、**C1Chart** の **Actions** プロパティです。**Action** オブジェクトには、インターフェースをカスタマイズできる複数のプロパティがあります。これらのプロパティはすべて、[プロパティ]ウィンドウと[Action コレクションエディター:Actions]を使用して、または XAML や Actions コレクションを使用したプログラムにより、デザイン時に設定または変更できます。

次の XAML は、エンドユーザーの操作を有効にするように **Actions** プロパティを設定する方法を示します。

XAML

```
<c1:C1Chart.Actions>
    <c1:ZoomAction />
    <c1:TranslateAction Modifiers="Shift" />
    <c1:ScaleAction Modifiers="Control" />
</c1:C1Chart.Actions>
```

次のコードは、C1Chart.Actions コレクションを通して **Actions** のプロパティをプログラムで設定する方法を示します。


C#

```
c1.Chart.Actions.Add(new ZoomAction());
```


次のリストは、サポートされているグラフ操作を示しています。

Chart for WPF/Silverlight


- 回転操作では、表示の角度を変更できます。この操作は、3D 効果を伴うグラフでのみ利用できます。**Rotate3DAction** クラスは、3D グラフの回転操作を表します (WPF のみ)。
- スケール操作は、選択した1つまたは複数の軸に沿ってグラフのスケールを増減させます。**ScaleAction** クラスは、スケール操作を表します。

 **注意:** **MinScale** プロパティが0の場合、ズームはグラフの軸に適用できません。**MinScale** プロパティは、軸に設定できる最小目盛を指定します。

- 変換操作では、プロット領域全体をスクロールして移動できるようになります。**TranslateAction** クラスは、変換操作を表します。

 **注意:** **Axis.Scale** プロパティが1を超える場合、軸の変換は利用できません。

- ズーム操作では、ユーザーは矩形領域を選択して表示できます。

 **注意:** **MinScale** プロパティが0の場合、ズームはグラフの軸に適用できません。**MinScale** プロパティは、軸に設定できる最小目盛を指定します。

スケール調整、変換、およびズームは、デカルト軸を持つグラフでのみ利用できます。

実行時のインタラクティブな回転は、3D グラフで利用できます。


Action オブジェクトは、操作時の動作のカスタマイズに役立つ一連のプロパティを提供します。

- **MouseButton** プロパティと**Modifiers** プロパティは、操作の実行を呼び出すマウスボタンとキー (Alt、Ctrl、Shift) の組み合わせを指定します。

3D グラフの回転の変更 (WPF のみ)

実行時に 3D グラフタイプの回転表示を変更するには、**Rotate3DAction** クラスを **Actions** コレクションに追加します。たとえば、マウスの中央ボタンでグラフを回転させるには、次の XAML コードを使用します。

```
<clchart:C1Chart.Actions>
<clchart:Rotate3DAction MouseButton="Middle" />
</clchart:C1Chart.Actions>
```

 **メモ:** このセクションの内容は、ComponentOne Studio for WPF にのみ適用されます。

2D でカルトグラフの実行時のインタラクティブ操作の実装

ズーム、スケール調整、および変換の操作は、指定のマウスボタンで呼び出されます。キーボード上の変更キー ([Alt]、[Ctrl]、または [Shift]) を押しながらの操作を指定することもできます。それらの操作は、**Actions** コレクションに配置する必要があります。次の XAML コードは、一連の操作を定義しています。

XAML

```
<clchart:C1Chart.Actions>
<!-- マウスの左ボタンを使用してデータをスクロール -->
<clchart:TranslateAction MouseButton="Left" />
<!-- [Ctrl]キーとマウスの左ボタンを使用してスケールを変更 -->
<clchart:ScaleAction MouseButton="Left" Modifiers="Ctrl"/>
<!-- [Shift]キーとマウスの左ボタンを使用して、選択された矩形領域をズーム表示 -->
<clchart:ZoomAction MouseButton="Left" Modifiers="Shift" />
</clchart:C1Chart.Actions>
```

各操作は、**Axis** のプロパティ (**Min**、**Max**、**Scale**、**MinScale**) と密接に関係しています。**Axis.Scale=1** のときは、その軸で変

換操作は利用できません。**MinScale** は、操作時に達成できるズームまたはスケールの制限を設定します。

C1Chart でズームする

C1Chart にズーム動作を追加するには、チャートの MouseWheel イベントでカスタムコードを使用します。

C#

```
private void chart_MouseWheel(object sender, MouseWheelEventArgs e)
{
    if (Keyboard.Modifiers == ModifierKeys.Control && e.Delta == -120)
    {
        chart.View.AxisX.Scale += .1;
        chart.View.AxisY.Scale += .1;
    }
    else if (Keyboard.Modifiers == ModifierKeys.Control && e.Delta == 120)
    {
        chart.View.AxisX.Scale -= .1;
        chart.View.AxisY.Scale -= .1;
    }
}
```

チャートをズームしながら移動できるようにするには、C1Chart の XAML に次のコードを追加します。

XAML

```
<clc:C1Chart x:Name="chart" MouseWheel="chart_MouseWheel" >
  <clc:C1Chart.Actions>
    <clc:TranslateAction MouseButton="Left" />
  </clc:C1Chart.Actions>
</clc:C1Chart>
```

バブルチャートをズームしながら拡大縮小する

バブルチャートをズームしながら拡大縮小するには、次のように **PlotElementLoaded** イベントでスケールを調整します。

C#

```
var ds = new BubbleSeries()
{
    XValuesSource = new double[] { 1, 2, 3, 4 },
    ValuesSource = new double[] { 1, 2, 3, 4 },
    SizeValuesSource = new double[] { 1, 2, 3, 4 },
};
ds.PlotElementLoaded += (s, e) =>
{
    var pe = (PlotElement)s;
    pe.RenderTransform = new ScaleTransform()
    {
        ScaleX = 1.0 / chart.View.AxisX.Scale,
        ScaleY = 1.0 / chart.View.AxisY.Scale
    };
};
```

Chart for WPF/Silverlight

```
pe.RenderTransformOrigin = new Point(0.5, 0.5);
};
chart.Data.Children.Add(ds);
chart.ChartType = ChartType.Bubble;
chart.Actions.Add(new TranslateAction());
chart.Actions.Add(new ScaleAction() { Modifiers = ModifierKeys.Control });
```

マーカーとラベル

ComponentOne Chart for WPF/Silverlight は、連結されたインタラクティブなマーカーやラベルの表示を特別にサポートしています。チャート内でマーカーを作成または表示する方法は1つではないため、必要に応じて適切な設定を行うことができるように、C1Chart コントロールのための拡張可能なオブジェクトモデルが提供されています。

このトピックでは、ChartPanelObject および ChartView.Layers コレクションを使用して、カスタマイズされたさまざまなマーカーやラベルをチャートに提供する方法について説明します。

チャートでチャートパネルを使用するには、そのパネルを ChartView の Layers コレクションに追加する必要があります。

XAML

```
<clchart:C1Chart x:Name="chart">
  <clchart:C1Chart.View>
    <clchart:ChartView>
      <clchart:ChartView.Layers>
        <clchart:ChartPanel >
          <!-- ChartPanelObjects -->
        </clchart:ChartPanel>
      </clchart:ChartView.Layers>
    </clchart:ChartView>
  </clchart:C1Chart.View>
</clchart:C1Chart>
```

ChartView.Layers コレクションを使用すれば、チャートパネルをいくつでも追加できます。各パネルには、任意の数の ChartPanelObject を置くことができ、これが基本的にマーカーを定義する UI 要素になります。ChartPanelObject の主なプロパティは次のとおりです。


- **Attach** – オブジェクトがデータポイントにアタッチされるか、それとも「スナップ」されるかを設定します。X 値、Y 値、その両方へのアタッチ、またはアタッチなしを設定できます。
- **Action** – マウス移動、マウスドラッグなどのユーザー操作に対する動作または動作なしを設定します。
- **DataPoint** – 初期データポイントを明示的に設定します。つまり、静的マーカーを作成します。

ChartPanelObject.Content プロパティは任意の UI 要素に設定できます。これにより、マーカーの外観を定義できると共に、データポイントへの連結も指定できます。また、**Alignment** プロパティを使用してマーカーの外観を定義することもできます。たとえば、位置を中央にしてマーカーを作成できます。それには、**HorizontalAlignment** プロパティを "center" に設定します。

次の XAML は、左下隅がデータ座標 $x=0, y=0$ にあるテキストラベルを定義します。

XAML

```
<cl:ChartPanelObject DataPoint="0,0" VerticalAlignment="Bottom">
  <TextBlock Text="ゼロ"/>
</cl:ChartPanelObject>
```

 **メモ:** 必ずしも両方の座標を指定する必要はありません。座標を double.NaN に設定した場合、要素は特定の x または y 座標を持たなくなります。

y=0 の水平マーカを作成できます。**HorizontalAlignment** プロパティを **Stretch** に設定すると、要素の幅がプロットエリアの幅いっぱいまで広がられます。


XAML

```
<!-- 水平線 -->
<cl:ChartPanelObject DataPoint="NaN,0" HorizontalAlignment="Stretch">
  <Border BorderBrush="Red" BorderThickness="0,2,0,0" Margin="0,-1,0,0" />
</cl:ChartPanelObject>
```

次のサンプルは、垂直マーカを作成します。

XAML

```
<!-- 垂直線 -->
<cl:ChartPanelObject DataPoint="0,NaN" VerticalAlignment="Stretch">
  <Border BorderBrush="Red" BorderThickness="2,0,0,0" Margin="-1,0,0,0" />
</cl:ChartPanelObject>
```

 **メモ:** チャートパネルオブジェクトは、主軸をサポートしています。補助軸の場合には、座標変換を行う必要があります。

シンプルな連結マーカ

5つのプロパティを設定するだけで、シンプルな連結マーカを簡単に作成できます。以下の XAML マークアップに、この例を示します。

XAML

```
<!-- シンプルな連結マーカ -->
<cl:ChartPanelObject x:Name="obj" Attach="DataX"
  Action="MouseMove"
  DataPoint="-1,-1"
  HorizontalAlignment="Center"
  VerticalAlignment="Top"
  Width="60" Height="50">
  <cl:ChartPanelObject.RenderTransform>
    <TranslateTransform Y="-50"/>
  </cl:ChartPanelObject.RenderTransform>
  <Grid DataContext="{Binding RelativeSource={x:Static
RelativeSource.Self},Path=Parent}" Opacity="0.8">
    <Path Data="M0.5,0.5 L23,0.5 23,23 11.61165,29.286408 0.5,23 z"
Stretch="Fill" Fill="#FFF1F1F1" Stroke="DarkGray" StrokeThickness="1"/>
    <StackPanel VerticalAlignment="Center" HorizontalAlignment="Center">
      <TextBlock Text="Value" Margin="2 0"/>
      <TextBlock x:Name="label" Text="{Binding DataPoint.Y, StringFormat=c2}"
FontWeight="Bold" Margin="2"/>
    </StackPanel>
  </Grid>
</cl:ChartPanelObject>
```

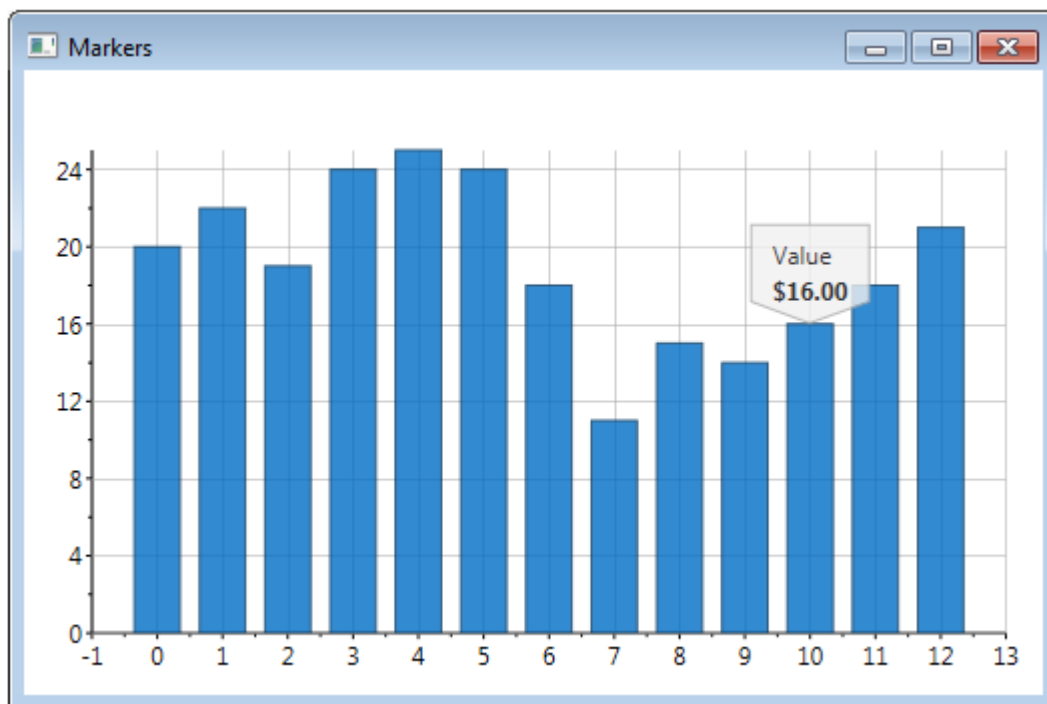
このマークアップでは、次のプロパティを設定しています。

- Attach = DataX
- Action = MouseMove

Chart for WPF/Silverlight

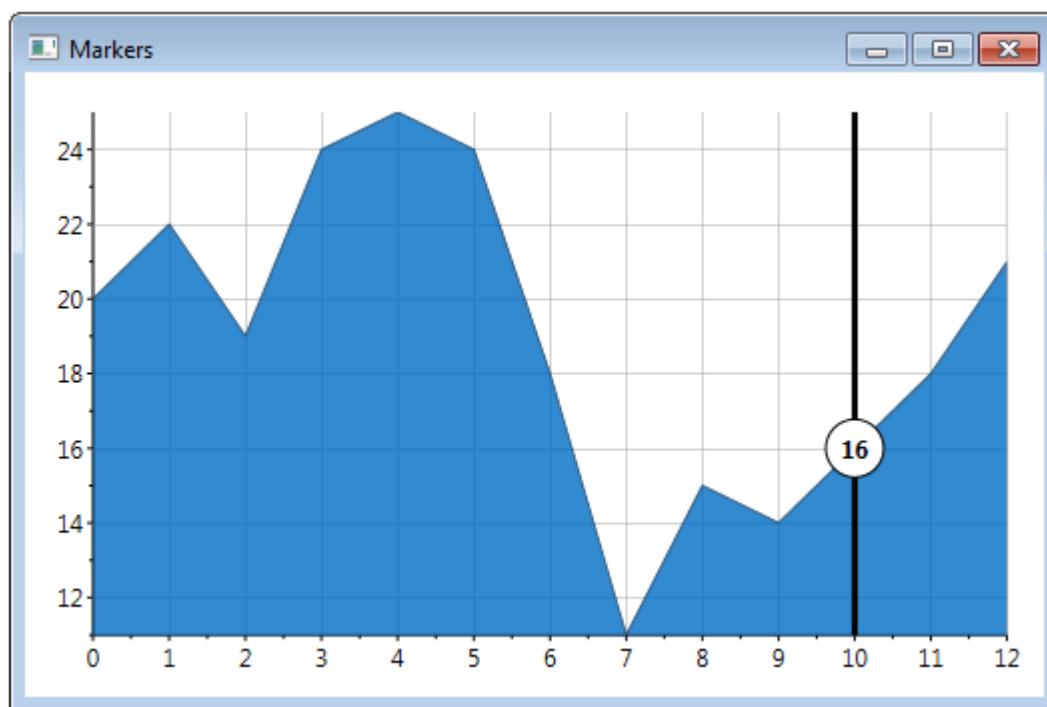
- DataPoint = -1,-1
- HorizontalAlignment = Center
- VerticalAlignment = Top

アプリケーションは、次の図のようになります。



線とドットのマーカー

面グラフや折れ線グラフでは、X 軸または Y 軸全体とデータポイントをマークするマーカーがほしい場合があります。次の図に、そのような線とドットのマーカーの例を示します。



このようなマーカーを作成するために設定するプロパティとして最も重要なものの1つは、**VerticalAlignment** プロパティで

す。このプロパティを "Stretch" に設定すると、マーカーがプロットの高さ全体に引き伸ばされて垂直線になります。マークアップで、次のプロパティを設定します。

- Attach = DataX
- Action = MouseMove
- DataPoint = -1, NaN
- HorizontalAlignment = Center
- VerticalContentAlignment = Stretch

データポイントの Y の値を NaN に設定していることに注目してください。このように設定すると、マーカーは特定のデータポイントにアタッチしないため、垂直方向いっぱい伸びた直線が引かれます。上の図の丸いラベルは、プロット要素上に配置される別の ChartPanelObject です。この DataPoint プロパティは NaN 以外の値に設定します。

この効果は、XAML マークアップを使用するだけで作成でき、コードは必要ありません。

XAML

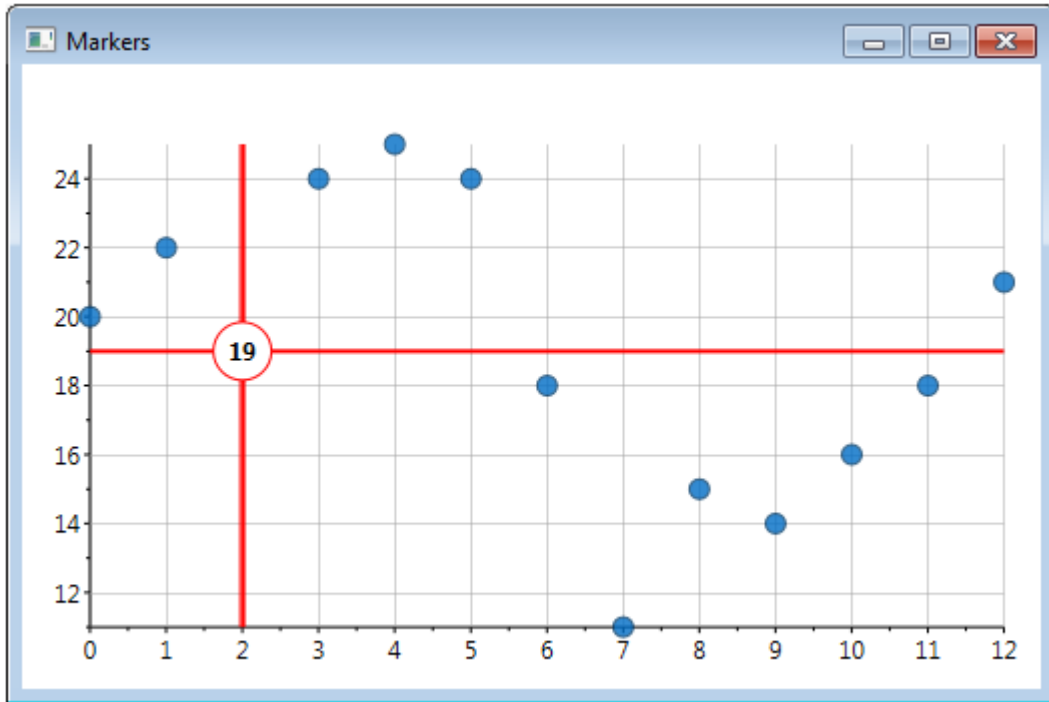
```
<!-- 垂直線とドットのマーカー -->
<cl:ChartPanelObject x:Name="vline"
    Attach="DataX"
    Action="MouseMove"
    DataPoint="-1, NaN"
    VerticalContentAlignment="Stretch"
    HorizontalAlignment="Center">
    <Border Background="Black" BorderBrush="Black" Padding="1" BorderThickness="1
0 0 0" />
</cl:ChartPanelObject>
<cl:ChartPanelObject x:Name="dot"
    Attach="DataX"
    Action="MouseMove"
    DataPoint="0.5,0.5"
    HorizontalAlignment="Center"
    VerticalAlignment="Center">
    <Grid DataContext="{Binding RelativeSource={x:Static
RelativeSource.Self},Path=Parent}">
        <Ellipse Fill="White" Stroke="Black" StrokeThickness="1" Width="30"
Height="30" />
        <TextBlock x:Name="label" Text="{Binding DataPoint.Y, StringFormat=n0}"
FontWeight="Bold" VerticalAlignment="Center" HorizontalAlignment="Center"/>
    </Grid>
</cl:ChartPanelObject>
```

上のマークアップだけで、線とドットのチャートマーカーを作成できます。

十字線マーカー

チャートによっては、自由に動かせる十字線タイプのマーカーを使用してデータポイントを強調したい場合があります。このトピックでは、線とドットのマーカーを元に、水平マーカーを追加して十字線マーカーを作成します。完成した十字線マーカー付きのチャートは、次の図のようになります。

Chart for WPF/Silverlight



以下の XAML でも、DataPoint は NaN に設定します。

XAML

```
<!-- 十字線 -->
<cl:ChartPanelObject x:Name="vline"
    Attach="None"
    Action="MouseMove"
    DataPoint="-1, NaN"
    VerticalContentAlignment="Stretch"
    HorizontalAlignment="Center">
    <Border Background="Red" BorderBrush="Red" Padding="1" BorderThickness="1 0 0
0" />
</cl:ChartPanelObject>
<cl:ChartPanelObject x:Name="hline"
    Attach="None"
    Action="MouseMove"
    DataPoint="NaN, -1"
    HorizontalContentAlignment="Stretch"
    VerticalAlignment="Center">
    <Border Background="Red" BorderBrush="Red" Padding="1" BorderThickness="0 1 0 0"
/>
</cl:ChartPanelObject>
<cl:ChartPanelObject x:Name="dot"
    Attach="None"
    Action="MouseMove"
    DataPoint="0.5,0.5"
    HorizontalAlignment="Center"
    VerticalAlignment="Center">
    <Grid DataContext="{Binding RelativeSource={x:Static
RelativeSource.Self},Path=Parent}">
        <Ellipse Fill="White" Stroke="Red" StrokeThickness="1" Width="30" Height="30"

```

```

/>
        <TextBlock x:Name="label" Text="{Binding DataPoint.Y, StringFormat=n0}"
        FontWeight="Bold" VerticalAlignment="Center" HorizontalAlignment="Center"/>
    </Grid>
</cl:ChartPanelObject>

```

コードでのマーカーの追加

ここまでのトピックでは、XAML マークアップを使用してマーカーを追加する方法を説明してきました。プロジェクトによっては、コードでマーカーを追加する必要があります。

最初に、新しい **ChartPanel** を作成する必要があります。

```

C#
var pnl = new ChartPanel();

```

新しい **ChartPanel** を追加したら、新しい **ChartPanelObject** を追加し、配置を設定します。

```

C#
var obj = new ChartPanelObject()
{
    HorizontalAlignment = HorizontalAlignment.Right,
    VerticalAlignment = VerticalAlignment.Bottom
};

```

次に、Border 要素を追加します。

```

C#
var bdr = new Border()
{
    Background = new SolidColorBrush(Colors.Green) { Opacity = 0.4 },
    BorderBrush = new SolidColorBrush(Colors.Green),
    BorderThickness = new Thickness(1, 1, 3, 3),
    CornerRadius = new CornerRadius(6, 6, 0, 6),
    Padding = new Thickness(3)
};

```

2つの **TextBlock** コントロールを含む **StackPanel** 要素を追加します。連結ソースは、追加した **ChartPanelObject** です。

```

C#
var sp = new StackPanel();
var tb1 = new TextBlock();
var bind1 = new Binding();
bind1.Source = obj;
bind1.StringFormat = "x={0:#.##}";
bind1.Path = new PropertyPath("DataPoint.X");
tb1.SetBinding(TextBlock.TextProperty, bind1);
var tb2 = new TextBlock();
var bind2 = new Binding();
bind2.Source = obj;
bind2.StringFormat = "y={0:#.##}";

```

Chart for WPF/Silverlight

```
bind2.Path = new PropertyPath("DataPoint.Y");
tb2.SetBinding(TextBlock.TextProperty, bind2);
sp.Children.Add(tb1);
sp.Children.Add(tb2);
bdr.Child = sp;
```

ChartPanelObject の **Content**、**DataPoint**、および **Action** プロパティを設定し、**ChartPanelObject** を **ChartPanel** に追加します。コードの最後の行は、レイヤのコレクションをチャートコントロールに追加しています。

C#

```
obj.Content = bdr;
obj.DataPoint = new Point();
obj.Action = ChartPanelAction.MouseMove;
pnl.Children.Add(obj);
chart.View.Layers.Add(pnl);
```

コードの最後の行で、**Attach** プロパティを設定する必要があります。

C#

```
obj.Attach = ChartPanelAttach.MouseMove;
    };
    }
}
```

このトピックのコードでは、マウスポインタに追従するチャートマーカーを作成しました。

コードでのラベルの更新

コードを使用して、フォームのラベル要素を更新することもできます。たとえば、チャートの外部に配置したラベルをマーカーの値で更新するようなマーカーを作成したい場合があります。それには、マーカーに対する **DataPointChanged** イベントを監視する必要があります。

次のコードは、マーカーのデータポイント値を取得し、その値をフォームの **TextBlock** に設定します。

C#

```
private void ChartPanelObject_DataPointChanged(object sender, EventArgs e)
{
    //コードで、マーカーからラベルを更新します
    var obj = (ChartPanelObject) sender;
    if (obj != null)
    {
        lbl.Text = obj.DataPoint.Y.ToString("c2");
    }
}
```

ChartPanel のマウス操作

ChartPanel は、マウス操作をサポートしています。**ChartPanelAction** 列挙体は、グラフパネルオブジェクトに対するアクションを指定します。**ChartPanelAction** 列挙体には、次のメンバが含まれます。

| メンバ名 | 説明 |
|----------------------|-----------------------|
| None | アクションなし。 |
| MouseMove | マウスポインタに追従します。 |
| LeftMouseButtonDrag | 左マウスボタンを使用してドラッグできます。 |
| RightMouseButtonDrag | 右マウスボタンを使用してドラッグできます。 |

Action プロパティを使用して、ドラッグ可能な要素またはマウスポインタに追従する要素を作成できます。たとえば、前のサンプルにアクションを追加して、ユーザーがマーカーを移動できるようにすることができます。

XAML

```
<!-- 垂直線 -->
<cl:ChartPanelObject DataPoint="0,NaN" VerticalAlignment="Stretch"
    Action="LeftMouseButtonDrag" >
    <Border BorderBrush="Red" BorderThickness="3,0,0,0"
Margin="-1.5,0,0,0" />
</cl:ChartPanelObject>
```

データ連結を使用すると、現在の座標を示すラベルを簡単に追加できます。

XAML

```
<!-- 座標ラベル付きの垂直線 -->
<cl:ChartPanelObject x:Name="xmarker" DataPoint="0,NaN"
VerticalAlignment="Stretch"
    Action="LeftMouseButtonDrag">
    <Border BorderBrush="Red" BorderThickness="3,0,0,0"
Margin="-1.5,0,0,0" >
    <TextBlock
        Text="{Binding RelativeSource={RelativeSource Self},
        Path=Parent.Parent.DataPoint.X, StringFormat='x=0.0;x=-0.0'}" />
    </Border>
</cl:ChartPanelObject>
```

Attach プロパティを使用すると、要素の位置を最も近いデータポイントにアタッチできます。一方の座標(X または Y)または両方の座標(XY)にアタッチできます。

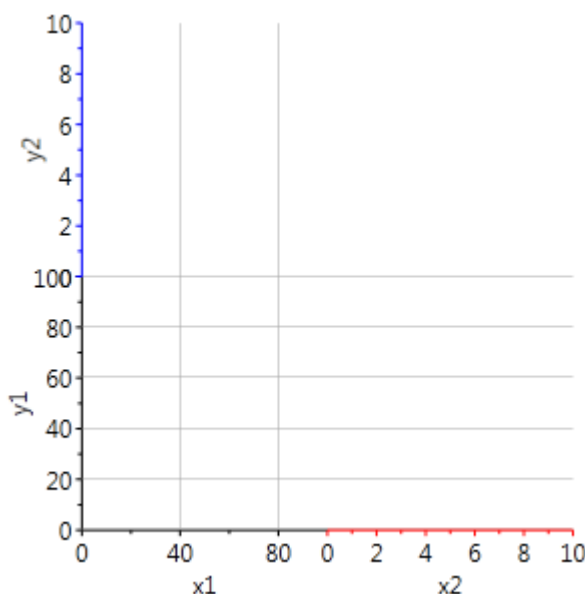
複数のプロット領域

データは、グラフのプロットエリアにプロットされます。プロットエリアは、軸で囲まれたプロット部分で、すべてのプロット要素(横棒、縦棒、線など)が含まれます。従来は1つのプロットエリアしか持つことができませんでしたが、新しく1つのグラフに複数のプロットエリアを持つことができるようになりました。

通常、プロットエリアは、**PlotAreaIndex** プロパティに基づいて自動的に作成されます。このプロパティはデフォルトで0です。この場合は、軸を追加しても新しいプロットエリアは作成されません。たとえば、単に軸は主 Y 軸の左側または主 X 軸の下側に追加されます。ただし、**PlotAreaIndex** = 1 と設定すると、新しい軸が主軸の同一線上に追加されます。X 軸の場合は補助軸が右側に、Y 軸の場合は補助軸が上側に表示されます。

次の例は、主軸の同一線上に追加された新しい軸を示します。

Chart for WPF/Silverlight



XAML

```
<cl:C1Chart x:Name="chart" >
  <cl:C1Chart.View>
    <cl:ChartView>
      <!-- 主軸 -->
      <cl:ChartView.AxisX>
        <cl:Axis Min="0" Max="100" Title="x1" />
      </cl:ChartView.AxisX>
      <cl:ChartView.AxisY>
        <cl:Axis Min="0" Max="100" Title="y1" />
      </cl:ChartView.AxisY>

      <!-- 主 X 軸の右側の補助軸 -->
      <cl:Axis x:Name="x2" Title="x2" PlotAreaIndex="1"
        AxisType="X" Min="0" Max="10" />

      <!-- 主 Y 軸の上側の補助軸 -->
      <cl:Axis x:Name="y2" Title="y2" PlotAreaIndex="1"
        AxisType="Y" Min="0" Max="10" />

    </cl:ChartView>
  </cl:C1Chart.View>
</cl:C1Chart>
```

データを追加するには、軸(**DataSeries.AxisX/AxisY**)の名前を指定する必要があります。このデータは補助軸に沿ってプロットされます。

プロットエリアのサイズ

PlotArea のサイズは、**PlotAreaCollection** クラスの **ColumnDefinitions** コレクションと **RowDefinitions** コレクションを使用して指定できます。この方法は、標準のグリッドコントロールの操作と同じです。最初のコレクションには、列属性(幅)が含まれます。2番目のコレクションは行(高さ)に使用されます。デフォルトでは、どのプロットエリアも同じ幅、同じ高さになります。

次の例は、プロットエリアのサイズをプログラムで指定する方法を示します。

C#

```
// 幅
// 最初のプロットエリアの幅はデフォルト(使用可能なスペースを占有)
chart.View.PlotAreas.ColumnDefinitions.Add(new PlotAreaColumnDefinition());
// 2番目のプロットエリアの幅は 100 ピクセルに固定
chart.View.PlotAreas.ColumnDefinitions.Add(new PlotAreaColumnDefinition()
    { Width= new GridLength(100) });
// 高さ
// 最初のプロットエリアの高さは1*
chart.View.PlotAreas.RowDefinitions.Add(new PlotAreaRowDefinition()
    { Height = new GridLength(1, GridUnitType.Star) });
// 2番目のプロットエリアの高さは2*
chart.View.PlotAreas.RowDefinitions.Add(new PlotAreaRowDefinition()
    { Height = new GridLength(2, GridUnitType.Star) });
```

プロットエリアの外観

PlotArea の外観は、**Background** プロパティと、プロットエリアの境界線の **Stroke/StrokeThickness** プロパティを使用して変更できます。プロットエリアは、行/列を使用して参照されます(グリッド内の要素と同様)。

次の例は、プロットエリアの外観を変更する方法を示します。

XAML

```
<c1:ChartView.PlotAreas>
  <!-- row=0 col=0 -->
  <c1:PlotArea Background="#10FF0000" Stroke="Red" />
  <!-- row=1 col=0 -->
  <c1:PlotArea Row="1" Background="#1000FF00" />
  <!-- row=0 col=1 -->
  <c1:PlotArea Column="1" Background="#100000FF" />
  <!-- row=1 col=1 -->
  <c1:PlotArea Row="1" Column="1" Background="#10FFFF00"
  Stroke="Yellow" />
</c1:ChartView.PlotAreas>
```

パフォーマンスの最適化

チャートの最適化の有効化

C1Chart は大量のデータを扱うことができますが、このことがパフォーマンスの問題につながる場合があります。SetOptimizationRadius()|tag=SetOptimizationRadius_Method メソッドを使用すると、このような問題を簡単に解決できます。このメソッドを使用すると、同じデータポイントが繰り返し処理される回数を減らすことができます。

次のコードは、このメソッドの例を示したものです。

C#

```
LineAreaOptions.SetOptimizationRadius(c1Chart1, 5);
```

レンダリングモード

C1Chart では、チャートのパフォーマンスを制御できるように、3つのレンダリングモードが提供されています。すべてのチャートタイプがサポートされるデフォルトレンダリングモードと、2つの高パフォーマンスレンダリングモードがあります。高パフォーマンスレンダリングモードを使用すると、チャートを高速にレンダリングできますが、いくつかの制限があります。

| レンダリングモード | 制限 |
|-----------|--|
| Default | デフォルトのレンダリングモード。すべてのチャートタイプがサポートされます。 |
| Fast | ビットマップレンダリングモード。高パフォーマンスのレンダリングモードです。現時点では、 折れ線グラフ と シンボルチャート のみがサポートされています。データポイントのラベル、ツールチップ、および PlotElementLoaded イベントは使用できません。 |
| Bitmap | ビットマップモード。高パフォーマンスのレンダリングモードです。現時点では、 折れ線グラフ と シンボルチャート のみがサポートされています。データポイントのラベル、ツールチップ、および PlotElementLoaded イベントは使用できません。 |

バッチ更新の実行

変更のたびにグラフを更新することなく、バッチ更新を実行できます。**BeginUpdate()** メソッドと **EndUpdate()** メソッドの間にコードを置きます。

VisualBasic

```
C1Chart1.BeginUpdate ()  
    ' グラフの変更や書式設定、データの追加など  
    ...  
C1Chart1.EndUpdate ()
```

C#

```
c1Chart1.BeginUpdate ();  
// グラフの変更や書式設定、データの追加など  
...  
c1Chart1.EndUpdate ();
```

設定されているチャートの最適化の無効化

チャートの最適化を設定後に無効にするには、次の手順に従います。

Visual Basic

```
LineAreaOptions.SetOptimizationRadius(c1Chart1, 2.0)
```

C#

```
LineAreaOptions.SetOptimizationRadius(c1Chart1, 2.0);
```

次のように、デフォルト値 NaN に設定することもできます。

Visual Basic

```
LineAreaOptions.SetOptimizationRadius(c1Chart1, double.NaN)
```

C#

```
LineAreaOptions.SetOptimizationRadius(c1Chart1, double.NaN);
```

関数のプロット

C1Chart には、関数をプロットするためのエンジンが組み込まれています。関数をプロットするための組み込みエンジンを使用するには、**C1.WPF.C1Chart.Extended.dll** または **C1.Silverlight.Chart.Extended.dll** への参照をプロジェクトに追加する必要があります。

さまざまなアプリケーションで、さまざまな種類の関数を使用されます。**C1Chart** には、多くのアプリケーションの作成に必要なさまざまな種類の関数が用意されています。

次の2つの種類の関数がサポートされています。

1. 1変数の陽関数

- 1変数の陽関数は、 $y=f(x)$ と記述されます（「**YFunctionSeries** クラス」を参照）。
- 具体例として、有理関数、一次関数、多項式関数、二次関数、対数関数、指数関数などがあります。
- これらの関数は、科学者やエンジニアによってよく使用され、さまざまな種類の財務、予測、パフォーマンス測定などのアプリケーションに利用できます。

2. パラメータ関数

- 関数は一対の方程式 ($y=f_1(t)$ と $x=f_2(t)$ など) によって定義されます。t は、関数 f1 と f2 の変数/座標です。
- パラメータ関数は、独立変数から成る個別の関数によって X 座標と Y 座標が定義されるという点で、特別な種類の関数です。
- パラメータ関数は、数学や工学（熱伝導、流体力学、電磁理論、惑星運動、相対性理論の諸相など）で、さまざまな状態を表すために使用されます。
- パラメータ関数の詳細については、「**ParametricFunctionSeries** クラス」を参照してください。

コード文字列による関数の定義

解釈コード文字列を関数クラス (**YFunctionSeries** または **ParametricFunctionSeries**) の関数の定義に使用すると、文字列はコンパイルされ、生成コードはアプリケーションに動的に組み込まれます。実行速度は、他のコンパイルされたコードと同じです。

単純な1変数陽関数の場合は、**YFunctionSeries** クラスオブジェクトが使用されます。このオブジェクトには、1つのコードプロパティ (**FunctionCode**) があります。YFunction オブジェクトの場合、独立変数は常に "x" であると仮定されます。

パラメータ関数の場合は、一対の方程式を **ParametricFunctionSeries** クラスオブジェクトを使って定義する必要があります。このオブジェクトには2つのプロパティがあり、各座標にそれぞれ1つのプロパティが対応します。プロパティ (**XFunctionCode** と **YFunctionCode**) は、独立変数を常に "t" と仮定したコードを受け付けます。

関数の値の計算

Chart for WPF/Silverlight

Parametric と YFunction の関数の値は、**CalculateValue()** メソッドを使用して計算できます。

次のコードは、CalculateValue() メソッドの例を示したものです。

```
C#  
  
class MySeries : FunctionSeries  
{  
    void SomeMethod()  
    {  
        CalculateValue(...);  
    }  
}
```

C1Chart の保存とエクスポート

以下のタスクは、グラフをさまざまな形式に保存およびエクスポートする方法を示します。

グラフを PDF 形式にエクスポートする

グラフをビットマップ画像にエクスポートし、C1Pdf ライブラリを使用して、この画像を含む PDF を作成するには、次のコードを使用します。

```
C#  
  
// グラフの画像をストリームに保存します  
MemoryStream ms = new MemoryStream();  
chart.SaveImage(ms, ImageFormat.Png);  
// ストリームから画像のインスタンスを作成します  
var img = System.Drawing.Image.FromStream(ms);  
// PDF文書を作成して保存します  
C1PdfDocument pdf = new C1PdfDocument();  
pdf.DrawImage(img, new System.Drawing.RectangleF(0, 0, img.Width, img.Height));  
pdf.Save("doc.pdf");  
  
c1Chart1.View.AxisX.IsTime = true;  
c1Chart1.View.AxisX.AnnoFormat = "MMM-dd";  
// 時間軸で MajorUnit=31 とすると、グラフは  
// 月の日数が一定でないことを考慮しつつ、  
// 各月の1日をマークします  
c1Chart1.View.AxisX.MajorUnit = 31;
```

グラフ画像のエクスポート

次のコード例のように、**RenderTargetBitmap** を使用して、グラフ画像をエクスポートできます。

VisualBasic

```
Dim bm As New RenderTargetBitmap(CInt(c1Chart1.ActualWidth),  
CInt(c1Chart1.ActualHeight), 96, 96, PixelFormats.[Default])  
bm.Render(c1Chart1)
```

```
Dim enc As New PngBitmapEncoder()
enc.Frames.Add(BitmapFrame.Create(bm))
Dim fs As New FileStream("chart.png", FileMode.Create)
enc.Save(fs)
```

C#

```
RenderTargetBitmap bm = new RenderTargetBitmap(
    (int)c1Chart1.ActualWidth, (int)c1Chart1.ActualHeight,
    96, 96, PixelFormats.Default);
bm.Render(c1Chart1);
PngBitmapEncoder enc = new PngBitmapEncoder();
enc.Frames.Add(BitmapFrame.Create(bm));
FileStream fs = new FileStream("chart.png", FileMode.Create);
enc.Save(fs);
```

C1Chart を .Png ファイルとして保存する

C1Chart を .Png ファイルとして保存するには、次のコードを使用します。


VisualBasic

```
' 画像をファイルに保存します
Using stm = System.IO.File.Create("chart.png")
    c1Chart1.SaveImage(stm, C1.WPF.C1Chart.Extended.ImageFormat.Png)
End Using
```

C#

```
// 画像をファイルに保存します
using (var stm = System.IO.File.Create("chart.png"))
{
    c1Chart1.SaveImage(stm, C1.WPF.C1Chart.Extended.ImageFormat.Png);
}
```

系列の生成

 **メモ:** このトピックのサンプルについては、ブログ投稿「Chart Automatic Series Generation (チャートの系列の自動生成) (MVVM)」を参照してください。

MVVM を使用している開発者は、ChartData オブジェクトの2つのプロパティ SeriesItemSource と SeriesItemTemplate を使用して、複数の系列をビューモデルで完全に生成することができます。

年ごとに異なるデータ系列をプロットしたいが、設計時には年数が不明な場合は、ビューモデルで年数を指定できます。

最初に、プロパティを ViewModel に連結するための要素について取り上げます。XAML マークアップとコードで要点を説明した後、「MVVM による系列の自動生成」トピックで、必要なマークアップとコードのすべてを説明します。

Chart for WPF/Silverlight

次の XAML マークアップでは、2つのプロパティが設定されています。

XAML

```
<cl:C1Chart.Data>
  <cl:ChartData SeriesItemsSource="{Binding SeriesDataCollection}">
    <cl:ChartData.SeriesItemTemplate>
      <DataTemplate>
        <cl:DataSeries Label="{Binding Year}" ValuesSource="{Binding
Values}" />
      </DataTemplate>
    </cl:ChartData.SeriesItemTemplate>
  </cl:ChartData>
</cl:C1Chart.Data>
<cl:C1ChartLegend DockPanel.Dock="Right" />
</cl:C1Chart>
```

ChartData オブジェクトで、SeriesItemsSource プロパティと SeriesItemTemplate プロパティの両方が設定されています。また、SeriesItemsSource は ViewModel に連結されており、SeriesItemTemplate の Label プロパティと ValuesSource プロパティも同様です。

ViewModel の2つのコードセクションに注目してみましょう。最初のセクションは、SeriesData の ObservableCollection を作成しています。

C#

```
public ObservableCollection<SeriesData> SeriesDataCollection
{
    get
    {
        if (_seriesDataCollection == null)
        {
            _seriesDataCollection = new ObservableCollection<SeriesData>();
            for (int i = 0; i < ViewModelData.NUM_SERIES; i++)
                _seriesDataCollection.Add(new SeriesData(2010 + i));
        }
        return _seriesDataCollection;
    }
}
```

2番目のコードセクションは、カスタムビジネスオブジェクトです。年とデータ値が含まれています。

C#

```
public class SeriesData : INotifyPropertyChanged
{
    int _year;
    double[] _values;
    public SeriesData(int year)
    {
        _year = year;
        _values = new double[ViewModelData.NUM_POINTS];
        for (int i = 0; i < ViewModelData.NUM_POINTS; i++)
            _values[i] = ViewModelData.Rnd.Next(0, 100);
    }
}
```



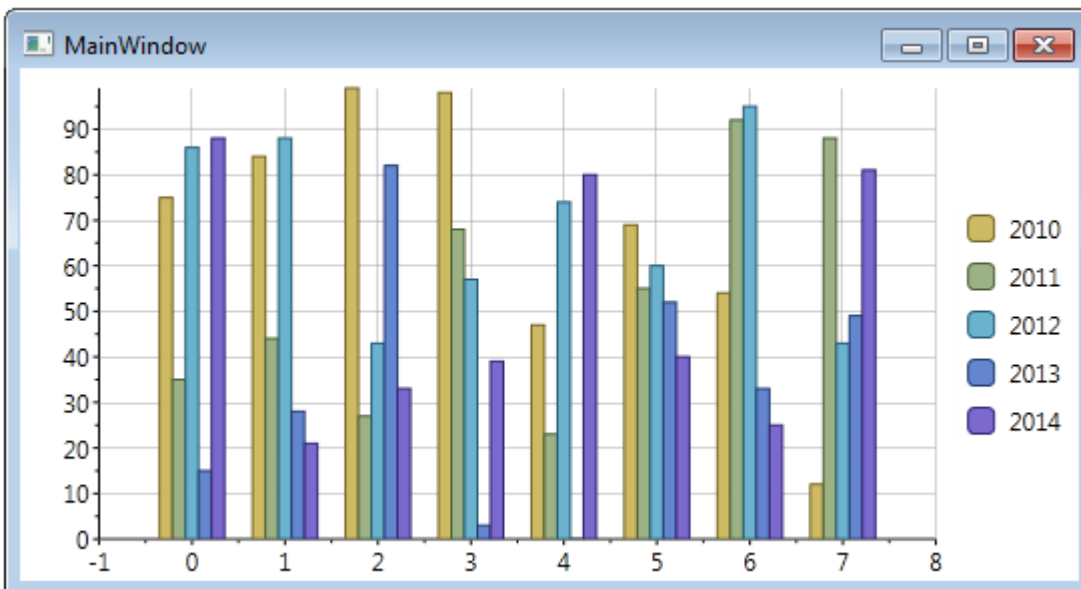
```

public int Year
{
    get { return _year; }
    set
    {
        if (_year != value)
        {
            _year = value;
            OnPropertyChanged("Year");
        }
    }
}

public double[] Values
{
    get { return _values; }
    set
    {
        if (_values != value)
        {
            _values = value;
            OnPropertyChanged("Values");
        }
    }
}

```

作成したプログラムまたはサンプルを実行すると、次の図のようになります。



MVVM による系列の自動生成

このトピックは、新しい Visual Studio プロジェクトが作成されており、プロジェクトに適切な参照が追加されていることを前提とします。

手順1: マークアップを作成します

Chart for WPF/Silverlight

次の XAML マークアップから始めます。

XAML

```
<c1:C1Chart Name="c1Chart1">
  <c1:C1Chart.Data>
    <c1:ChartData SeriesItemsSource="{Binding SeriesDataCollection}">
      <c1:ChartData.SeriesItemTemplate>
        <DataTemplate>
          <c1:DataSeries Label="{Binding Year}" ValuesSource="{Binding
Values}" />
        </DataTemplate>
      </c1:ChartData.SeriesItemTemplate>
    </c1:ChartData>
  </c1:C1Chart.Data>
  <c1:C1ChartLegend DockPanel.Dock="Right" />
</c1:C1Chart>
```

ChartData オブジェクトで SeriesItemsSource と SeriesItemTemplate が設定され、それぞれ値が連結されています。

手順2:ビューモデルを作成します

次に、プロジェクトのビューモデルを作成する必要があります。

- プロジェクト名を右クリックし、**[追加]**→**[新しい項目]**を選択します。
- **[コードファイル]**を選択し、"ViewModel.cs" と名前を付け、**[OK]**をクリックします。

次のコードをコードファイルに追加してビューモデルを作成します。

C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.ComponentModel;
using System.Collections.ObjectModel;
namespace ChartAutomaticSeries
{
    public static class ViewModelData
    {
        public static int NUM_SERIES = 5;
        public static int NUM_POINTS = 8;
        public static Random Rnd = new Random();
        private static ChartModelData _data;
        public static ChartModelData ChartData
        {
            get
            {
                if (_data == null)
                {
                    _data = new ChartModelData();
                }
                return _data;
            }
        }
    }
}
```

```

    }
}
public class ChartModelData
{
    public ObservableCollection<SeriesData> SeriesDataCollection
    {
        get
        {
            if (_seriesDataCollection == null)
            {
                _seriesDataCollection = new ObservableCollection<SeriesData>();
                for (int i = 0; i < ViewModelData.NUM_SERIES; i++)
                    _seriesDataCollection.Add(new SeriesData(2010 + i));
            }
            return _seriesDataCollection;
        }
    }
    private ObservableCollection<SeriesData> _seriesDataCollection;
}
public class SeriesData : INotifyPropertyChanged
{
    int _year;
    double[] _values;
    public SeriesData(int year)
    {
        _year = year;
        _values = new double[ViewModelData.NUM_POINTS];
        for (int i = 0; i < ViewModelData.NUM_POINTS; i++)
            _values[i] = ViewModelData.Rnd.Next(0, 100);
    }
    public int Year
    {
        get { return _year; }
        set
        {
            if (_year != value)
            {
                _year = value;
                OnPropertyChanged("Year");
            }
        }
    }
    public double[] Values
    {
        get { return _values; }
        set
        {
            if (_values != value)
            {
                _values = value;
                OnPropertyChanged("Values");
            }
        }
    }
}

```

```
    }
    #region INotifyPropertyChanged Members
    public event PropertyChangedEventHandler PropertyChanged;
    private void OnPropertyChanged(string propertyName)
    {
        if (PropertyChanged != null)
        {
            PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
        }
    }
    #endregion
}
}
```

手順3:コードを追加します

MainWindow.xaml ファイルに切り替えます。ページを右クリックし、コンテキストメニューから[コードの表示]を選択します。既存のコードを次のように編集します。

```
C#
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
        this.DataContext = new ChartModelData();
    }
}
}
```

系列の作成

グラフのデータ系列は、手動または自動で作成できます。

AutoGenerateSeries プロパティは、系列が自動的に作成されるかどうかを指定します。デフォルトでは **AutoGenerateSeries** プロパティは null であり、データ系列は **Children** コレクションが空の場合にしか作成されません。系列が作成されている間に、グラフでは **Data.ItemsSource** (または **C1Chart.DataContext**) コレクションの各要素が分析され、サポートされている型 (**numeric**、**DateTime**) のプロパティごとに系列が作成されます。

系列作成のプロセスを制御するには、プロットするプロパティを指定できる **Bindings** プロパティを使用します。**Bindings** プロパティの詳細については、「[データ系列のバインディング](#)」を参照してください。

レイアウトおよび外観

以下のトピックでは、C1Chart コントロールの外観をカスタマイズする方法について詳しく説明します。グラフがより明確に、およびよりプロフェッショナルな形に見えるようにするには、グラフの要素をカスタマイズすることができます。

グラフのリソースキー

組み込みテーマとリソースには、いくつかのリソースキーが実装されています。これらのキーはブラシ、枠線、その他の要素などがあり、カスタマイズすれば独自の外観を表現できます。テーマをカスタマイズすると、明示的に指定されていないリソース

キーはデフォルトに戻ります。以下のトピックでは、付属のリソースキーとそれらの説明を示します。

下の表は、Chart コントロールとその要素(グラフ領域、プロット領域、軸、凡例など)のリソースキーについて説明しています。

グラフのリソースキー

| リソースキー | 説明 |
|--------------------------|-----------------------------|
| C1Chart_Foreground_Color | C1Chart の前景色を表します。 |
| C1Chart_Background_Color | C1Chart の背景色を表します。 |
| C1Chart_Background_Brush | C1Chart の背景ブラシを表します。 |
| C1Chart_Foreground_Brush | C1Chart の前景ブラシを表します。 |
| C1Chart_Border_Brush | C1Chart の枠線ブラシを表します。 |
| C1Chart_Border_Thickness | C1Chart の枠線の太さ(4辺すべて)を表します。 |
| C1Chart_CornerRadius | グラフの角の丸み(4つの角すべて)を表します。 |
| C1Chart_Padding | C1Chart のパディングを表します。 |
| C1Chart_Margin | C1Chart の余白を表します。 |

凡例のリソースキー

| リソースキー | 説明 |
|--------------------------------|--------------------------------------|
| C1Chart_LegendBackground_Brush | C1Chart の凡例の背景ブラシを表します。 |
| C1Chart_LegendForeground_Brush | C1Chart コントロールの凡例の前景ブラシを表します。 |
| C1Chart_LegendBorder_Brush | C1Chart コントロールの凡例の枠線ブラシを表します。 |
| C1Chart_LegendBorder_Thickness | C1Chart コントロールの凡例の枠線の太さ(4辺すべて)を表します。 |
| C1Chart_Legend_CornerRadius | 凡例の角の丸み(4つの角すべて)を表します。 |

グラフ領域のリソースキー

| リソースキー | 説明 |
|-----------------------------------|--|
| C1Chart_ChartAreaBackground_Brush | ChartArea の背景ブラシを表します。 |
| C1Chart_ChartAreaForeground_Brush | マウスポインタが置かれているときの ChartArea の前景ブラシを表します。 |
| C1Chart_ChartAreaBorder_Brush | ChartArea の枠線ブラシを表します。 |
| C1Chart_ChartAreaBorder_Thickness | ChartArea の枠線の太さを表します。 |
| C1Chart_ChartArea_CornerRadius | ChartArea の角の丸み(4つの角すべて)を表します。 |
| C1Chart_ChartArea_Padding | ChartAreas のパディングを表します。 |

プロット領域のリソースキー

| リソースキー | 説明 |
|----------------------------------|-----------------------|
| C1Chart_PlotAreaBackground_Brush | PlotArea の背景ブラシを表します。 |

プロット要素のカスタムパレットのキー

Chart for WPF/Silverlight

| リソースキー | 説明 |
|-----------------------|-----------------------|
| C1Chart_CustomPalette | プロット要素のカスタムパレットを表します。 |

軸のキー

| リソースキー | 説明 |
|-----------------------------------|--------------------------------|
| C1Chart_AxisMajorGridStroke_Brush | AxisMajorGridStroke のブラシを表します。 |
| C1Chart_AxisMinorGridStroke_Brush | AxisMinorGridStroke のブラシを表します。 |

グラフスタイル

プロット要素は、グラフの外観を簡単に制御できるように WPF/Silverlight スタイルをサポートします。


MouseOver スタイル

次の例は、PlotElement の **Stroke** プロパティを *Black* に設定するスタイルを作成する方法を示します。

これは、要素上にマウスポインタを置くと外観が変更される MouseOver スタイルのサンプルです。

XAML

```
<Window.Resources>
  ...
  <Style x:Key="mouseOver" TargetType="{x:Type clc:PlotElement}">
    <!-- デフォルトの黒色のアウトライン -->
    <Setter Property="Stroke" Value="Black" />
    <Style.Triggers>
      <!-- 要素上にマウスを置くと太い赤色のアウトラインに変わる -->
      <Trigger Property="IsMouseOver" Value="true">
        <Setter Property="Stroke" Value="Red" />
        <Setter Property="StrokeThickness" Value="3" />
        <Setter Property="Canvas.ZIndex" Value="1" />
      </Trigger>
    </Style.Triggers>
  </Style>
</Window.Resources>
```

 **メモ:** x:Key を使用してスタイルを割り当てずに、スタイルの TargetType を PlotElement タイプに設定すると、このスタイルが両方の PlotElement 要素に適用されます。

使用可能なデータ系列にこの MouseOver スタイルに適用するには、次のように **SymbolStyle** プロパティを設定します。

XAML

```
<clc:DataSeries ... SymbolStyle="{StaticResource mouseOver}"/>
```

MouseOver スタイル

次の例は、PlotElement の **Stroke** プロパティを *Black* に設定するスタイルを作成する方法を示します。

これは、要素上にマウスポインタを置くと外観が変更される MouseOver スタイルのサンプルです。

XAML

```

<Window.Resources>
    ...
    <Style x:Key="mouseover" TargetType="{x:Type clc:PlotElement}">
        <!-- デフォルトの黒色のアウトライン -->
        <Setter Property="Stroke" Value="Black" />
        <Style.Triggers>
            <!-- 要素上にマウスを置くと太い赤色のアウトラインに変わる -->
            <Trigger Property="IsMouseOver" Value="true">
                <Setter Property="Stroke" Value="Red" />
                <Setter Property="StrokeThickness" Value="3" />
                <Setter Property="Canvas.ZIndex" Value="1" />
            </Trigger>
        </Style.Triggers>
    </Style>
</Window.Resources>

```

メモ: x:Key を使用してスタイルを割り当てずに、スタイルの TargetType を PlotElement タイプに設定すると、このスタイルが両方の PlotElement 要素に適用されます。

使用可能なデータ系列にこの MouseOver スタイルに適用するには、次のように **SymbolStyle** プロパティを設定します。

XAML

```

<clc:DataSeries ... SymbolStyle="{StaticResource mouseOver}"/>

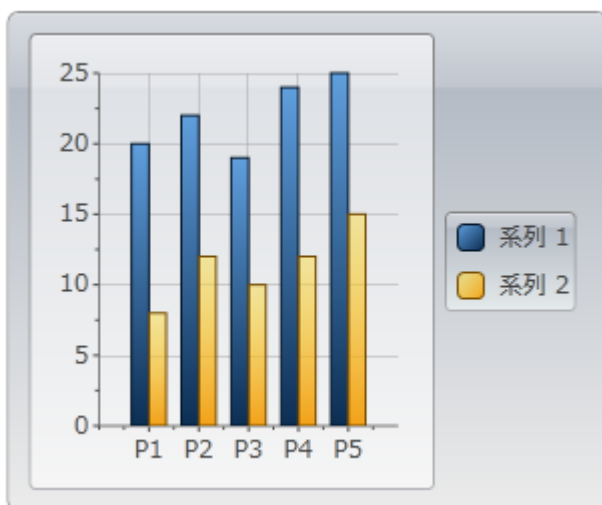
```

テーマ

Chart for WPF/Silverlight には **Office 2003**、**Vista**、**Office 2007** などいくつかのテーマが組み込まれており、これらを使ってグラフの外観をカスタマイズできます。以下に、組み込みテーマについて説明および図示します。

Office2007Black テーマ

これは、**Office 2007** の「黒」スタイルに基づくデフォルトのテーマです。グラフは濃い灰色、強調表示はオレンジ色で表示されます。



XAML の場合

Chart for WPF/Silverlight

グラフで **Office2007Black** テーマを明示的に定義するには、次に示すように、Theme XAML を <c1chart:C1Chart> タグに追加します。

WPF

XAML

```
<c1chart:C1Chart Name="c1Chart1" Theme="{DynamicResource {ComponentResourceKey  
TypeInTargetAssembly=c1chart:C1Chart, ResourceId=Office2007Black}}">
```

Silverlight

XAML

```
<c1:C1Chart Name="c1Chart1" Theme="Office2007Black">
```

コードの場合

グラフで **Office2007Black** テーマを明示的に定義するには、次のコードをプロジェクトに追加します。

WPF

Visual Basic

```
C1Chart1.Theme = TryCast(C1Chart1.TryFindResource( _  
    New ComponentResourceKey(GetType(C1Chart), "Office2007Black")), _  
    ResourceDictionary)
```

C#

```
c1Chart1.Theme = c1Chart1.TryFindResource(  
    new ComponentResourceKey(typeof(C1.WPF.C1Chart.C1Chart),  
    "Office2007Black")) as ResourceDictionary;
```

Silverlight

Visual Basic

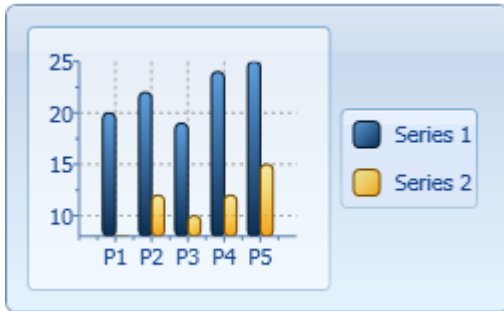
```
C1Chart1.Theme = ChartTheme.Office2007Black
```

C#

```
C1Chart1.Theme = ChartTheme.Office2007Black
```

Office2007Blue テーマ

このテーマは、**Office 2007 の「青」**スタイルに基づいています。グラフは青色、強調表示はオレンジ色で表示されます。



XAML の場合

グラフで **Office2007Blue** テーマを明示的に定義するには、次に示すように、Theme XAML を `<c1chart:C1Chart>` タグに追加します。

WPF

XAML

```
<c1chart:C1Chart Name="c1Chart1" Theme="{DynamicResource {ComponentResourceKey
TypeInTargetAssembly=c1chart:C1Chart, ResourceId=Office2007Blue}}">
```

Silverlight

XAML

```
<c1:C1Chart Name="c1Chart1" Theme="Office2007Blue">
```

コードの場合

グラフで **Office2007Blue** テーマを明示的に定義するには、次のコードをプロジェクトに追加します。

WPF

Visual Basic

```
C1Chart1.Theme = TryCast(C1Chart1.TryFindResource(
    New ComponentResourceKey(GetType(C1Chart), " Office2007Blue")),
    ResourceDictionary)
```

C#

```
C1Chart1.Theme = c1Chart1.TryFindResource(
    new ComponentResourceKey(typeof(C1.WPF.C1Chart.C1Chart),
    "Office2007Blue")) as ResourceDictionary;
```

Silverlight

Visual Basic

```
C1Chart1.Theme = ChartTheme.Office2007Blue
```

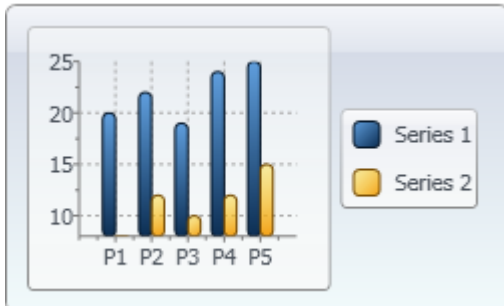
C#

Chart for WPF/Silverlight

```
C1Chart1.Theme = ChartTheme.Office2007Blue;
```

Office2007Silver テーマ

このテーマは、Office 2007 の「シルバー」スタイルに基づいています。グラフはシルバー、強調表示はオレンジ色で表示されます。



XAML の場合

グラフで Office2007Silver テーマを明示的に定義するには、次に示すように、Theme XAML を <c1chart:C1Chart> タグに追加します。

WPF

XAML

```
<c1chart:C1Chart Name="c1Chart1" Theme="{DynamicResource {ComponentResourceKey  
TypeInTargetAssembly=c1chart:C1Chart, ResourceId=Office2007Silver}}">
```

Silverlight

XAML

```
<c1:C1Chart Name="c1Chart1" Theme="Office2007Silver">
```

コードの場合

グラフで Office2007Silver テーマを明示的に定義するには、次のコードをプロジェクトに追加します。

WPF

Visual Basic

```
C1Chart1.Theme = TryCast(C1Chart1.TryFindResource( _  
    New ComponentResourceKey(GetType(C1Chart), "Office2007Silver")), _  
    ResourceDictionary)
```

C#

```
c1Chart1.Theme = c1Chart1.TryFindResource(  
    new ComponentResourceKey(typeof(C1.WPF.C1Chart.C1Chart),  
    "Office2007Silver")) as ResourceDictionary;
```

Silverlight

Visual Basic

```
C1Chart1.Theme = ChartTheme.Office2007Silver
```

C#

```
c1Chart1.Theme = ChartTheme.Office2007Silver;
```

Vista テーマ

このテーマは、**Vista** スタイルに基づいています。グラフは青緑色、強調表示は青色で表示されます。

XAML の場合

グラフで **Vista** テーマを明示的に定義するには、次に示すように、Theme XAML を <c1chart:C1Chart> タグに追加します。

WPF

XAML

```
<c1chart:C1Chart Name="c1Chart1" Theme="{DynamicResource {ComponentResourceKey
TypeInTargetAssembly=c1chart:C1Chart, ResourceId=Vista}}">
```

Silverlight

XAML

```
<c1:C1Chart Name="c1Chart1" Theme="Vista">
```

コードの場合

グラフで **Vista** テーマを明示的に定義するには、次のコードをプロジェクトに追加します。

WPF

Visual Basic

```
C1Chart1.Theme = TryCast(C1Chart1.TryFindResource( _
    New ComponentResourceKey(GetType(C1Chart), "Vista"), _
    ResourceDictionary)
```

C#

```
c1Chart1.Theme = c1Chart1.TryFindResource(
    new ComponentResourceKey(typeof(C1.WPF.C1Chart.C1Chart),
    "Vista")) as ResourceDictionary;
```

Silverlight

Visual Basic

```
C1Chart1.Theme = ChartTheme.Vista
```

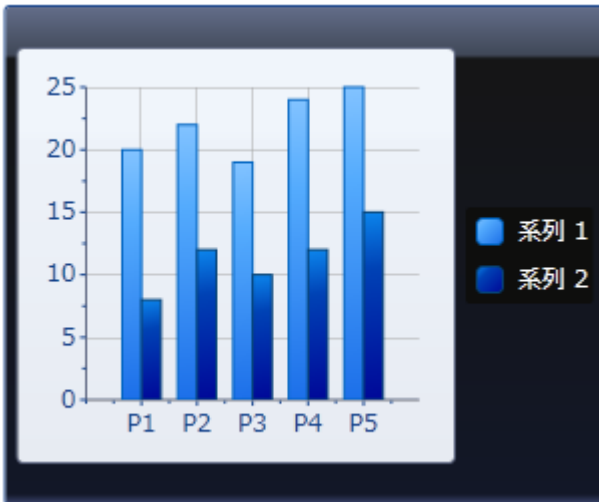
Chart for WPF/Silverlight

C#

```
c1Chart1.Theme = ChartTheme.Visata;
```

MediaPlayer テーマ

このテーマは、**Windows Media Player** スタイルに基づいています。グラフは黒色、強調表示は青色で表示されます。



XAML の場合

グラフで **MediaPlayer** テーマを明示的に定義するには、次に示すように、Theme XAML を `<c1chart:C1Chart>` タグに追加します。

WPF

XAML

```
<c1chart:C1Chart Name="c1Chart1" Theme="{DynamicResource {ComponentResourceKey  
TypeInTargetAssembly=c1chart:C1Chart, ResourceId=MediaPlayer}}">
```

Silverlight

XAML

```
<c1:C1Chart Name="c1Chart1"  
Theme="MediaPlayer">
```

コードの場合

グラフで **MediaPlayer** テーマを明示的に定義するには、次のコードをプロジェクトに追加します。

WPF

Visual Basic

```
C1Chart1.Theme = TryCast(C1Chart1.TryFindResource( _  
    New ComponentResourceKey(GetType(C1Chart), "MediaPlayer")), _  
    ResourceDictionary)
```

C#

```
c1Chart1.Theme = c1Chart1.TryFindResource(
    new ComponentResourceKey(typeof(C1.WPF.C1Chart.C1Chart),
        "MediaPlayer")) as ResourceDictionary;
```

Silverlight

Visual Basic

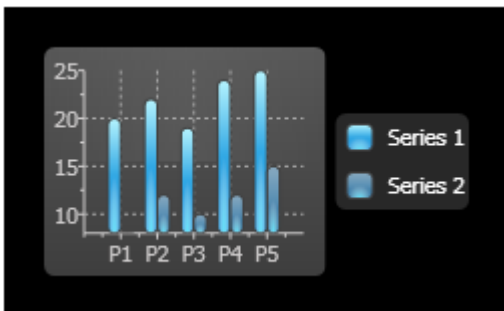
```
C1Chart1.Theme = ChartTheme.MediaPlayer
```

C#

```
c1Chart1.Theme = ChartTheme.MediaPlayer;
```

DuskBlue テーマ

このテーマでは、グラフがチャコールグレー、強調表示が明るい青色とオレンジ色で表示されます。



XAML の場合

グラフで **DuskBlue** テーマを明示的に定義するには、次に示すように、Theme XAML を タグに追加します。

WPF

XAML

```
<c1chart:C1Chart Name="c1Chart1" Theme="{DynamicResource {ComponentResourceKey
TypeInTargetAssembly=c1chart:C1Chart, ResourceId=DuskBlue}}">
```

Silverlight

XAML

```
<c1:C1Chart Name="c1Chart1"
Theme="DuskBlue">
```

コードの場合

グラフで **DuskBlue** テーマを明示的に定義するには、次のコードをプロジェクトに追加します。

WPF

Chart for WPF/Silverlight

Visual Basic

```
C1Chart1.Theme = TryCast(C1Chart1.TryFindResource( _  
    New ComponentResourceKey(GetType(C1Chart), "DuskBlue"), _  
    ResourceDictionary)
```

C#

```
c1Chart1.Theme = c1Chart1.TryFindResource(  
    new ComponentResourceKey(typeof(C1.WPF.C1Chart.C1Chart),  
    "DuskBlue")) as ResourceDictionary;
```

Silverlight

Visual Basic

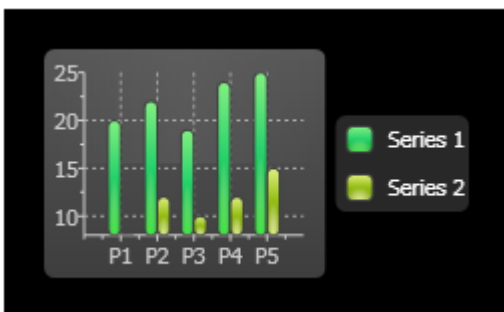
```
C1Chart1.Theme = ChartTheme.DuskBlue
```

C#

```
c1Chart1.Theme = ChartTheme.DuskBlue;
```

DuskGreen テーマ

このテーマでは、グラフがチャコールグレー、強調表示が明るい緑色と紫色で表示されます。



XAML の場合

グラフで **DuskGreen** テーマを明示的に定義するには、次に示すように、Theme XAML を <c1chart:C1Chart> タグに追加します。

WPF

XAML

```
<c1chart:C1Chart Name="c1Chart1" Theme="{DynamicResource {ComponentResourceKey  
TypeInTargetAssembly=c1chart:C1Chart, ResourceId=DuskGreen}} ">
```

Silverlight

XAML

```
<c1:C1Chart Name="c1Chart1"  
Theme="DuskGreen">
```

コードの場合

グラフで **DuskGreen** テーマを明示的に定義するには、次のコードをプロジェクトに追加します。

WPF

VisualBasic

```
C1Chart1.Theme = TryCast(C1Chart1.TryFindResource(
    New ComponentResourceKey(GetType(C1Chart), "DuskGreen"),
    ResourceDictionary))
```

C#

```
c1Chart1.Theme = c1Chart1.TryFindResource(
    new ComponentResourceKey(typeof(C1.WPF.C1Chart.C1Chart),
    " DuskGreen")) as ResourceDictionary;
```

Silverlight

Visual Basic

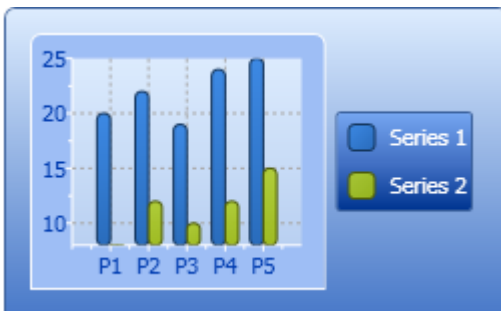
```
C1Chart1.Theme = ChartTheme.DuskGreen
```

C#

```
c1Chart1.Theme = ChartTheme.DuskGreen;
```

Office2003Blue テーマ

このテーマは、**Office 2003** の「青」スタイルに基づいています。グラフは中間色、強調表示は青色とオレンジ色で表示されます。



XAML の場合

グラフで **Office2003Blue** テーマを明示的に定義するには、次に示すように、Theme XAML を <c1chart:C1Chart> タグに追加します。

WPF

XAML

```
<c1chart:C1Chart Name="c1Chart1" Theme="{DynamicResource {ComponentResourceKey
TypeInTargetAssembly=c1chart:C1Chart, ResourceId=Office2003Blue}}">
```

Chart for WPF/Silverlight

Silverlight

XAML

```
<c1:C1Chart Name="c1Chart1" Theme="Office2003Blue">
```

コードの場合

WPF

Visual Basic

```
C1Chart1.Theme = TryCast(C1Chart1.TryFindResource( _  
    New ComponentResourceKey(GetType(C1Chart), "Office2003Blue")), _  
    ResourceDictionary)
```

C#

```
c1Chart1.Theme = c1Chart1.TryFindResource(  
    new ComponentResourceKey(typeof(C1.WPF.C1Chart.C1Chart),  
    "Office2003Blue")) as ResourceDictionary;
```

Silverlight

Visual Basic

```
C1Chart1.Theme = ChartTheme.Office2003Blue
```

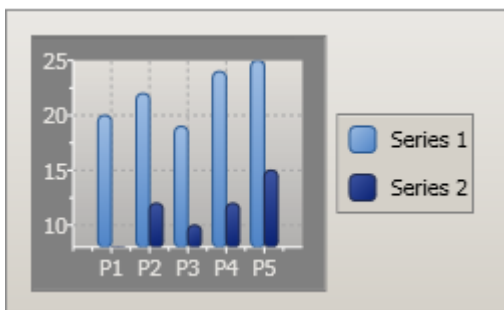
C#

```
c1Chart1.Theme = ChartTheme.Office2003Blue;
```

グラフで **Office2003Blue** テーマを明示的に定義するには、次のコードをプロジェクトに追加します。

Office2003Classic テーマ

このテーマは、**Office 2003** の「クラシック」スタイルに基づいています。グラフは灰色、強調表示はスレート色で表示されます。



XAML の場合

グラフで **Office2003Classic** テーマを明示的に定義するには、次に示すように、Theme XAML を `<c1chart:C1Chart>` タグに追加します。

WPF

XAML

```
<c1chart:C1Chart Name="c1Chart1" Theme="{DynamicResource {ComponentResourceKey
TypeInTargetAssembly=c1chart:C1Chart, ResourceId=Office2003Classic}">
```

Silverlight

XAML

```
<c1:C1Chart Name="c1Chart1"
Theme="Office2003Classic">
```

コードの場合

グラフで **Office2003Classic** テーマを明示的に定義するには、次のコードをプロジェクトに追加します。

WPF

Visual Basic

```
C1Chart1.Theme = TryCast(C1Chart1.TryFindResource( _
    New ComponentResourceKey(GetType(C1Chart), "Office2003Classic"), _
    ResourceDictionary)
```

C#

```
c1Chart1.Theme = c1Chart1.TryFindResource(
    new ComponentResourceKey(typeof(C1.WPF.C1Chart.C1Chart),
    "Office2003Classic")) as ResourceDictionary;
```

Silverlight

Visual Basic

```
C1Chart1.Theme = ChartTheme.Office2003Classic
```

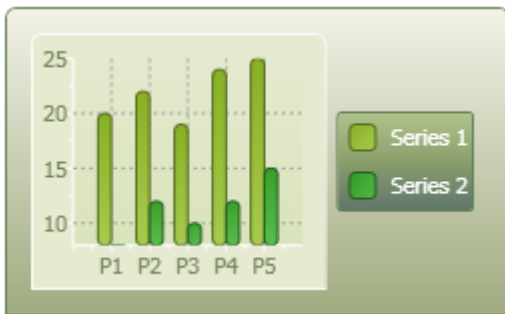
C#

```
c1Chart1.Theme = ChartTheme.Office2003Classic;
```

Office2003Olive テーマ

このテーマは、**Office 2003** の「**オリーブ**」スタイルに基づいています。グラフは中間色、強調表示はオリーブグリーンとオレンジで表示されます。

Chart for WPF/Silverlight



XAML の場合

グラフで **Office2003Olive** テーマを明示的に定義するには、次に示すように、Theme XAML を `<c1chart:C1Chart>` タグに追加します。

WPF

XAML

```
<c1chart:C1Chart Name="c1Chart1" Theme="{DynamicResource {ComponentResourceKey  
TypeInTargetAssembly=c1chart:C1Chart, ResourceId=Office2003Olive}}">
```

Silverlight

XAML

```
<c1:C1Chart Name="c1Chart1"  
Theme="Office2003Olive">
```

コードの場合

グラフで **Office2003Olive** テーマを明示的に定義するには、次のコードをプロジェクトに追加します。

WPF

Visual Basic

```
C1Chart1.Theme = TryCast(C1Chart1.TryFindResource( _  
    New ComponentResourceKey(GetType(C1Chart), "Office2003Olive"), _  
    ResourceDictionary)
```

```
c1Chart1.Theme = c1Chart1.TryFindResource(  
    new ComponentResourceKey(typeof(C1.WPF.C1Chart.C1Chart),  
    "Office2003Olive")) as ResourceDictionary;
```

Silverlight

Visual Basic

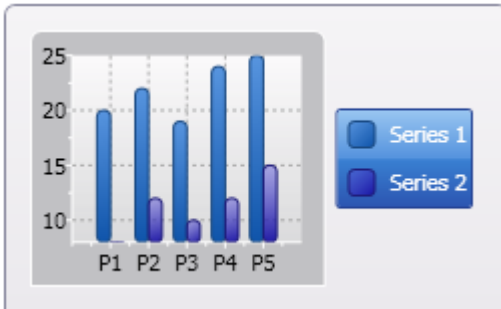
```
C1Chart1.Theme = ChartTheme.Office2003Olive
```

C#

```
c1Chart1.Theme = ChartTheme.Office2003Olive;
```

Office2003Royale テーマ

このテーマは、Office 2003 の「ロイヤル」スタイルに似ています。グラフはシルバー、強調表示は青色で表示されます。



XAML の場合

グラフで Office2003Royale テーマを明示的に定義するには、次に示すように、Theme XAML を <c1chart:C1Chart> タグに追加します。

WPF

XAML

```
<c1chart:C1Chart Name="c1Chart1" Theme="{DynamicResource {ComponentResourceKey
TypeInTargetAssembly=c1chart:C1Chart, ResourceId=Office2003Royale}}">
```

Silverlight

XAML

```
<c1:C1Chart Name="c1Chart1" Theme="Office2003Royale">
```

コードの場合

グラフで Office2003Royale テーマを明示的に定義するには、次のコードをプロジェクトに追加します。

WPF

Visual Basic

```
C1Chart1.Theme = TryCast(C1Chart1.TryFindResource( _
    New ComponentResourceKey(GetType(C1Chart), "Office2003Royale")), _
    ResourceDictionary)
```

C#

```
c1Chart1.Theme = c1Chart1.TryFindResource(
    new ComponentResourceKey(typeof(C1.WPF.C1Chart.C1Chart),
    " Office2003Royale")) as ResourceDictionary;
```

Silverlight

Chart for WPF/Silverlight

Visual Basic

```
C1Chart1.Theme = ChartTheme.Office2003Royale)
```

C#

```
c1Chart1.Theme = ChartTheme.Office2003Royale;
```

Office2003Silver テーマ

このテーマは、**Office 2003** の「シルバー」スタイルに基づいています。グラフはシルバー、強調表示は灰色とオレンジ色で表示されます。



XAML の場合

グラフで **Office2003Silver** テーマを明示的に定義するには、次に示すように、Theme XAML を <c1chart:C1Chart> タグに追加します。

WPF

XAML

```
<c1chart:C1Chart Name="c1Chart1" Theme="{DynamicResource {ComponentResourceKey  
TypeInTargetAssembly=c1chart:C1Chart, ResourceId=Office2003Silver}}">
```

Silverlight

XAML

```
<c1:C1Chart Name="c1Chart1"  
Theme="Office2003Silver">
```

コードの場合

グラフで **Office2003Royale** テーマを明示的に定義するには、次のコードをプロジェクトに追加します。

WPF

VisualBasic

```
C1Chart1.Theme = TryCast(C1Chart1.TryFindResource( _  
    New ComponentResourceKey(GetType(C1Chart), "Office2003Silver")), _  
    ResourceDictionary)
```

C#

```
c1Chart1.Theme = c1Chart1.TryFindResource(
    new ComponentResourceKey(typeof(C1.WPF.C1Chart.C1Chart),
        "Office2003Silver")) as ResourceDictionary;
```

Silverlight

VisualBasic

```
C1Chart1.Theme = ChartTheme.Office2003Silver)
```

C#

```
c1Chart1.Theme = ChartTheme.Office2003Silver;
```

データ系列のカラーパレット

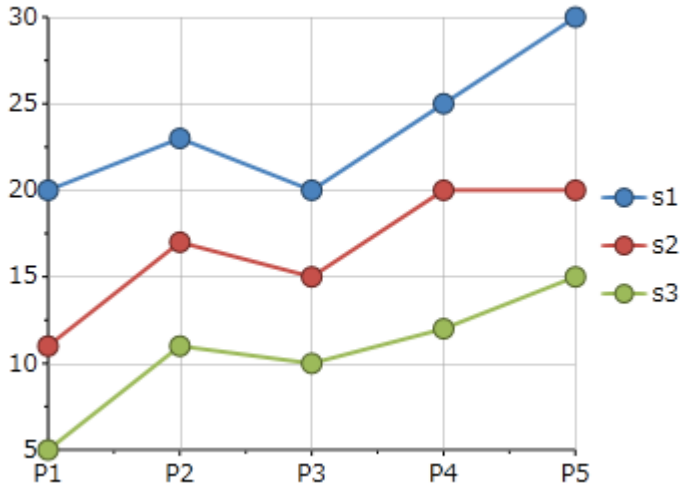
Palette プロパティを使用して、データ系列の配色を選択できます。デフォルトでは、**C1Chart** は **ColorGeneration.Default** の設定を使用します。残りのオプションは、Microsoft Office のカラーテーマに類似しています。

データ系列で利用可能な配色を以下に示します。

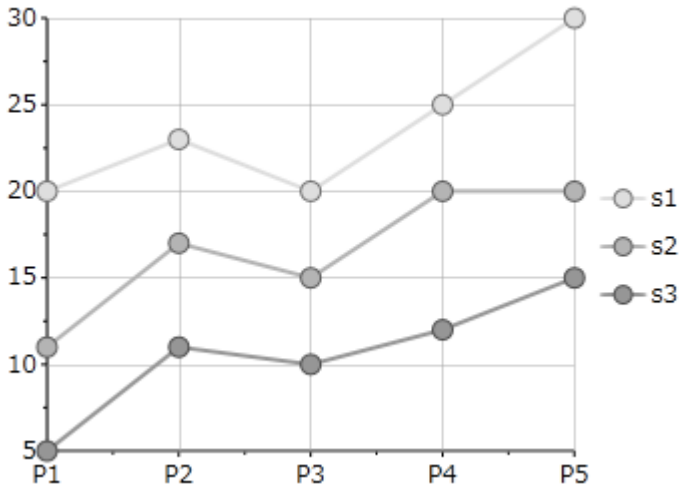
| 色生成設定 | 説明またはプレビュー | | | | | | | | | | | | | | | | | | | | | | | | |
|--------|---|--------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|---|----|----|----|----|
| デフォルト | C1Chart.ColorGeneration を "Default" に設定すると、グラフにテーマが設定されている場合はテーマパレットが使用され、設定されていない場合は[ひらめき]パレットが適用されます。 | | | | | | | | | | | | | | | | | | | | | | | | |
| 標準 | <table border="1"> <caption>Chart Data (Standard Palette)</caption> <thead> <tr> <th>Series</th> <th>P1</th> <th>P2</th> <th>P3</th> <th>P4</th> <th>P5</th> </tr> </thead> <tbody> <tr> <td>s1</td> <td>20</td> <td>23</td> <td>20</td> <td>25</td> <td>30</td> </tr> <tr> <td>s2</td> <td>11</td> <td>17</td> <td>15</td> <td>20</td> <td>20</td> </tr> <tr> <td>s3</td> <td>5</td> <td>11</td> <td>10</td> <td>12</td> <td>15</td> </tr> </tbody> </table> | Series | P1 | P2 | P3 | P4 | P5 | s1 | 20 | 23 | 20 | 25 | 30 | s2 | 11 | 17 | 15 | 20 | 20 | s3 | 5 | 11 | 10 | 12 | 15 |
| Series | P1 | P2 | P3 | P4 | P5 | | | | | | | | | | | | | | | | | | | | |
| s1 | 20 | 23 | 20 | 25 | 30 | | | | | | | | | | | | | | | | | | | | |
| s2 | 11 | 17 | 15 | 20 | 20 | | | | | | | | | | | | | | | | | | | | |
| s3 | 5 | 11 | 10 | 12 | 15 | | | | | | | | | | | | | | | | | | | | |

Chart for WPF/Silverlight

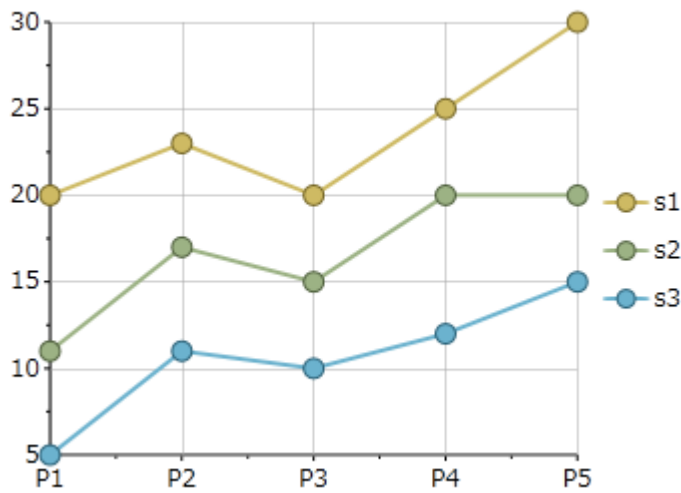
オフィス



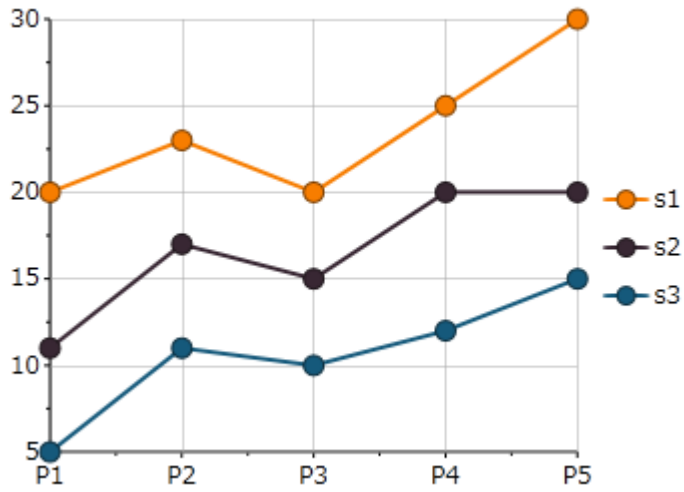
GrayScale



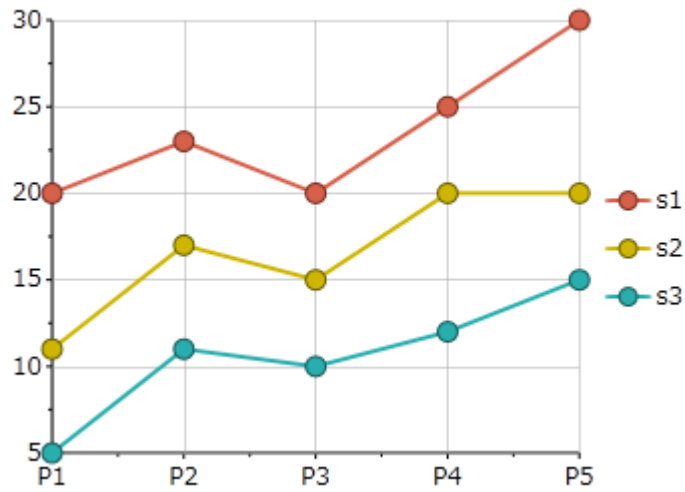
ひらめき



シック



クール



ビジネス

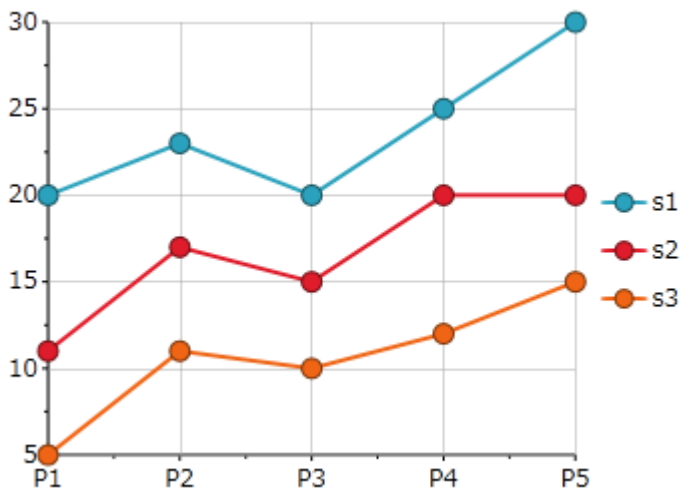
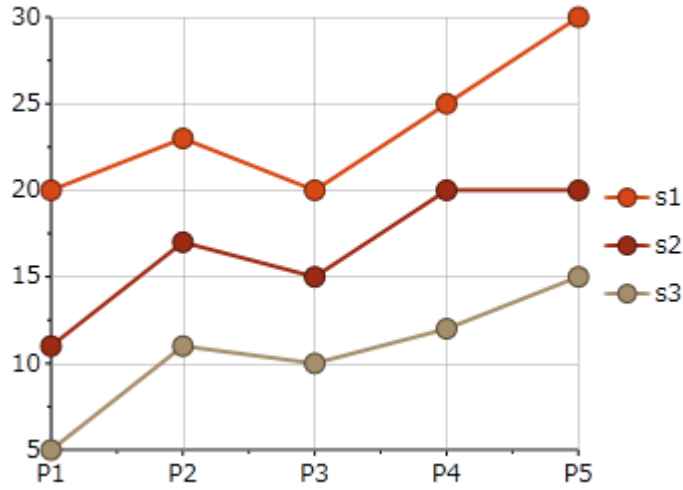
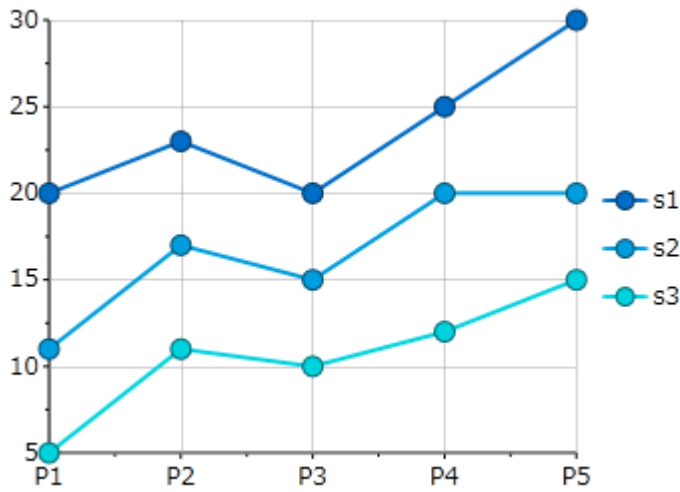


Chart for WPF/Silverlight

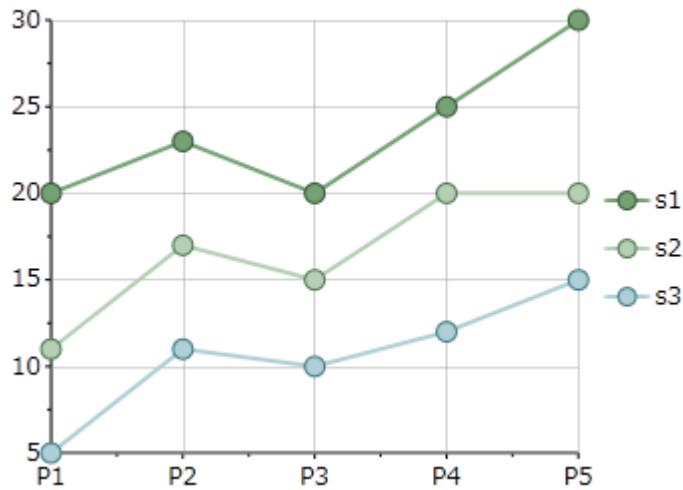
ジャパネ
スク



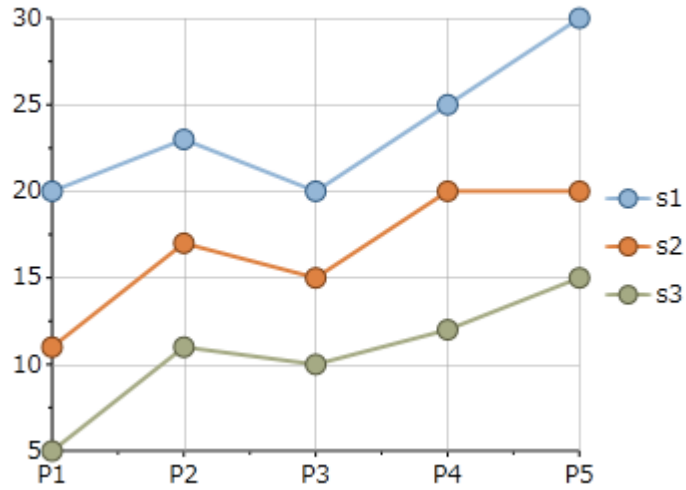
リゾート



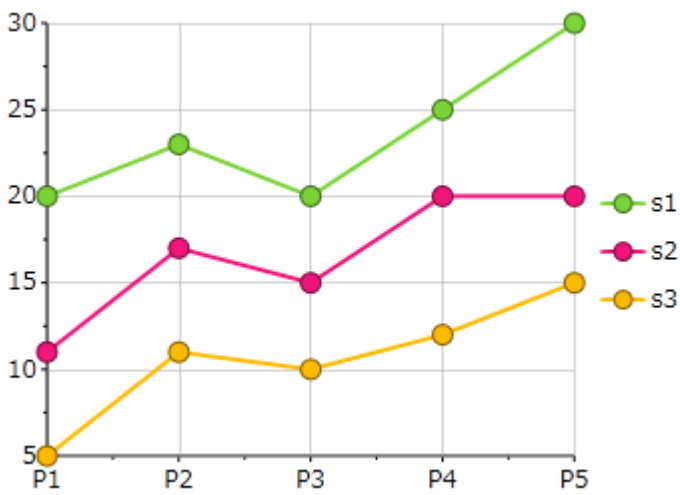
エコロ
ジー



デザート



メトロ



モジュール

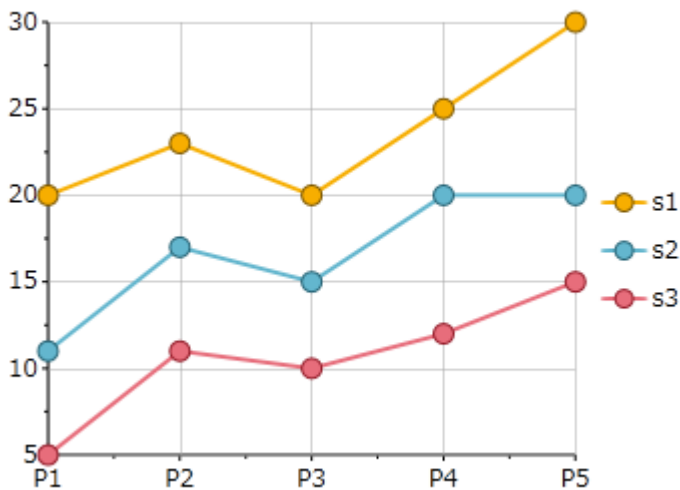
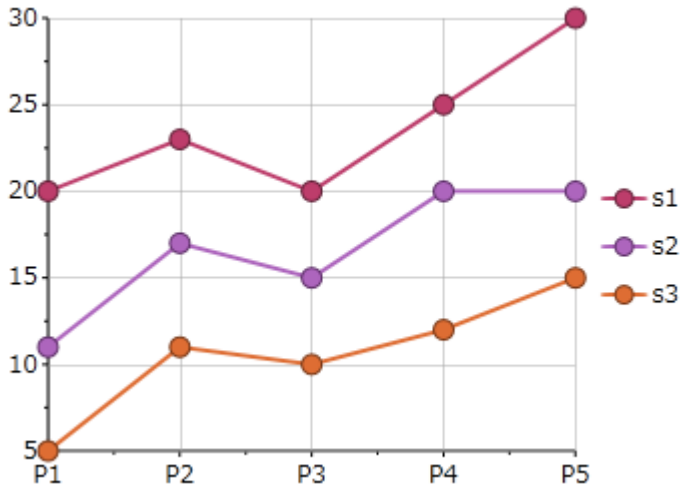
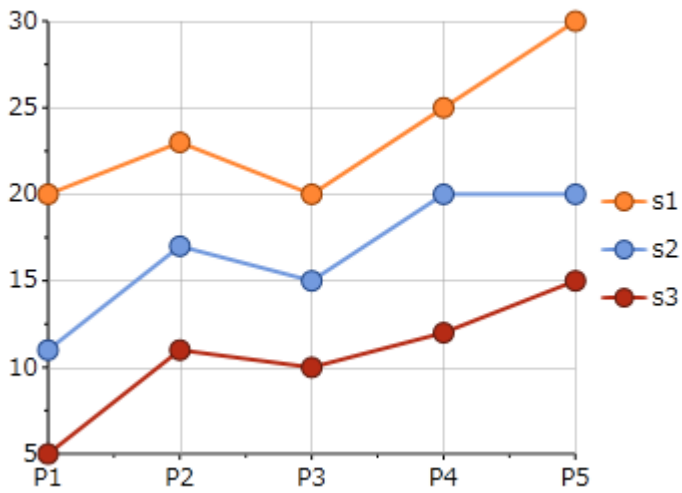


Chart for WPF/Silverlight

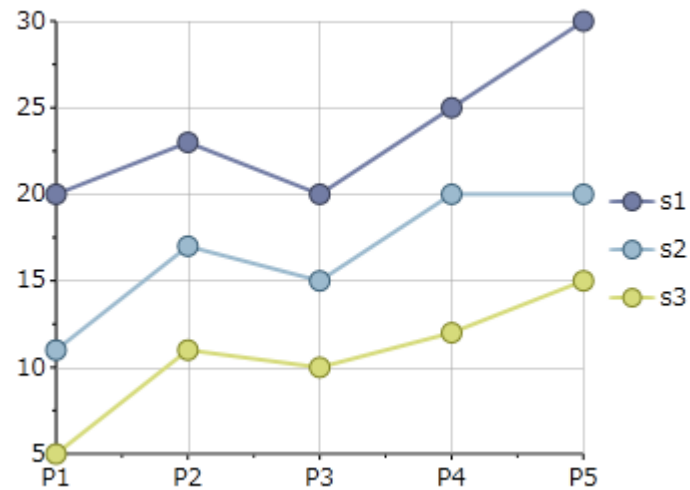
キュート



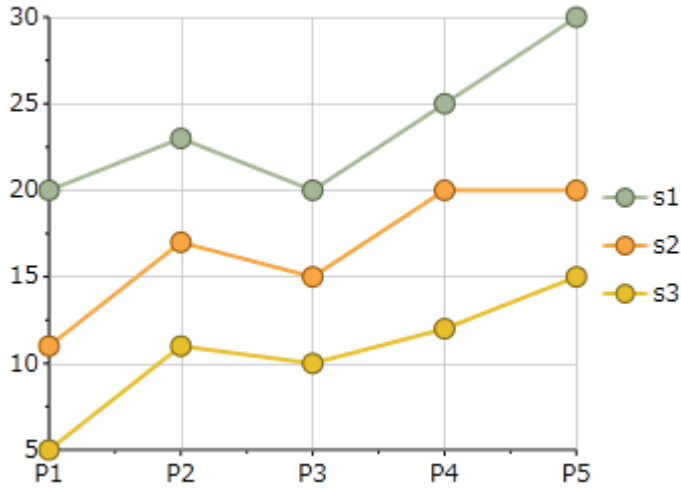
スパイス



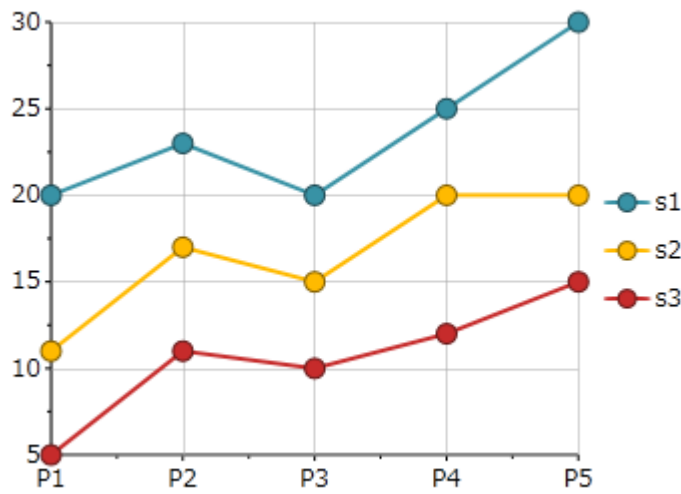
アース



ペーパー



フレッシュ



テクノ

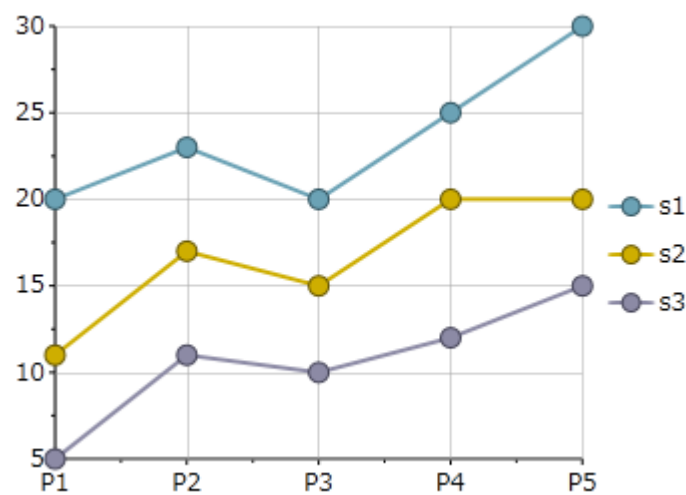
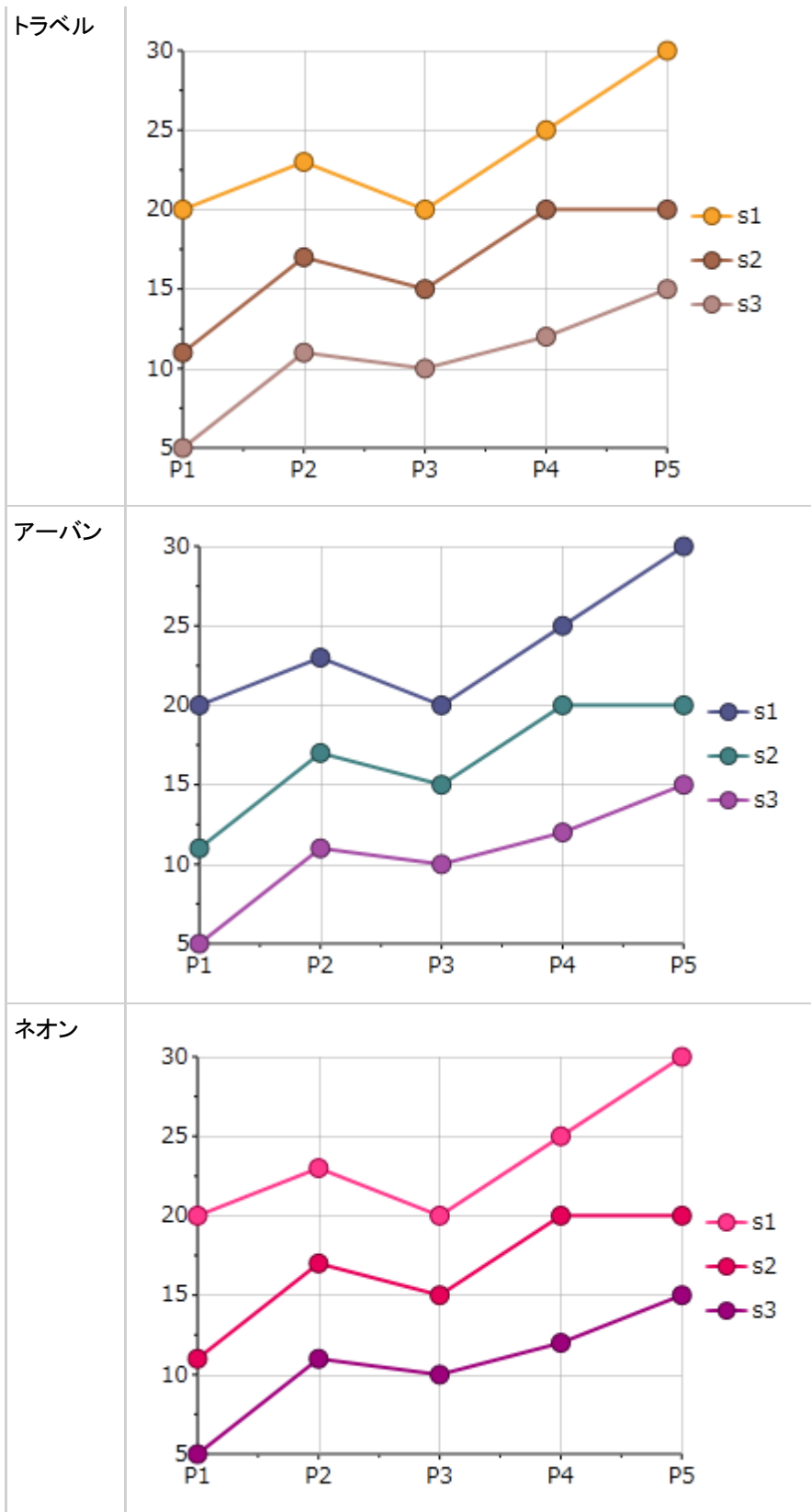


Chart for WPF/Silverlight



プロット要素の色を変更する

棒や円(グラフの種類に応じて)などプロット要素に割り当てられている色を変更するには、Paletteプロパティをいずれかの所定のカラーパレットに変更する、または以下のように、カスタムのパレットを作成することができます。

C#

```
Brush[] customBrushes = new Brush[2] { Brushes.Blue, Brushes.Orange };
c1Chart1.CustomPalette = customBrushes;
```

グラフの書式設定

前のセクションで紹介した**Theme**を使用すると、チャートの外観をすばやく簡単に選択できます。**Theme** プロパティと**Palette** プロパティでは、数多くの組み込みオプションが提供されています。これらは、開発者がほとんど労力をかけることなく素晴らしい結果を得ることができるように、細心の注意を払って開発されています。

ほとんどのアプリケーションの場合、開発者は、アプリケーションに必要と考える外観に最も近い **Theme** プロパティと **Palette** プロパティの設定の組み合わせを選択し、必要に応じていくつかの項目をカスタマイズします。カスタマイズの対象となる項目は、次のとおりです。

- 軸タイトル**: 軸タイトルは UIElement オブジェクトです。直接、完全に自由にカスタマイズできます。「[単純なグラフ](#)」、「[時系列グラフ](#)」、および「[散布グラフ](#)」のチャートサンプルでは、**TextElement** オブジェクトを使用していますが、**Border** オブジェクトや **Grid** オブジェクトなどのパネルを含むさまざまな要素を使用できます。軸タイトルの詳細については、「[軸のタイトル](#)」を参照してください。
- 軸**: 「[単純なグラフ](#)」、「[時系列グラフ](#)」、および「[散布グラフ](#)」のチャートサンプルでは、軸スケール、アニメーション角度、軸のスケール、注釈の角度、注釈の書式をカスタマイズする方法を示しています。これらはすべて、**AxisX** プロパティと**AxisY** プロパティによって公開される **Axis** オブジェクトからアクセスできます。**C1Chart** の軸の詳細については、「[軸](#)」を参照してください。
C1Chart コントロールには、両方の軸に沿って注釈をどのように表示するかを決定する、一般的な Font プロパティ (**FontFamily**、**FontSize** など)があります。注釈の外観をさらに細かく制御する必要がある場合は、**Axis** オブジェクトによって公開されている **AnnoTemplate** プロパティを使用して、さらに注釈をカスタマイズできます。
- グリッド線**: グリッド線は **Axis** のプロパティによって制御されます。主グリッド線用のプロパティと副グリッド線用のプロパティがあります (**MajorGridStrokeThickness**、**MinorGridStrokeThickness** など)。グリッド線の詳細については、「[軸のグリッド線](#)」を参照してください。
- 目盛りマーク**: 目盛りマークも **Axis** のプロパティによって制御されます。大目盛り用のプロパティと小目盛り用のプロパティがあります (**MajorTickStroke**、**MajorTickThickness**、**MinorTickStroke**、**MinorTickThickness** など)。目盛りマークの詳細については、「[軸の目盛り記号](#)」を参照してください。

XAML の要素

ComponentOne Studio をインストールすると、いくつかの補助的な XAML 要素がインストールされます。これらの要素にはテンプレートやテーマが含まれており、**ComponentOne Studio** のインストールディレクトリ下 (¥Misc¥Xaml¥WPF ¥C1WPFChart または ¥Misc¥Xaml¥Silverlight¥C1SilverlightChart) に格納されています。これらの要素をプロジェクトに組み込むことにより、たとえば、付属の **Office 2007** テーマに基づいて独自のテーマを作成できます。これらの要素のいくつかを表す組み込みテーマの詳細については、「[テーマ](#)」を参照してください。

付属の補助的な XAML 要素

以下の補助的な XAML 要素が **Chart for WPF/Silverlight** に付属しています。下の表には、それらの要素が置かれるフォルダの位置も記載されています。

| 要素 | フォルダ | 説明 |
|-----------------|--------|---------------------------------|
| ChartTypes.xaml | | すべての利用可能なグラフタイプのテンプレートを指定します。 |
| default.xaml | Themes | Default テーマのテンプレートを指定します。 |
| DuskBlue.xaml | Themes | Dusk Blue テーマのテンプレートを指定します。 |
| generic.xaml | Themes | グラフの各種スタイルと初期スタイルのテンプレートを指定します。 |
| Grayscale.xaml | Themes | Grayscale テーマのテンプレートを指定します。 |

| | | |
|------------------------|--------|---------------------------------------|
| Legend.xaml | Themes | Legend のテンプレートを指定します。 |
| MediaPlayer.xaml | Themes | Media Player テーマのテンプレートを指定します。 |
| Office2003Blue.xaml | Themes | Office 2003 Blue テーマのテンプレートを指定します。 |
| Office2003Classic.xaml | Themes | Office 2003 Classic テーマのテンプレートを指定します。 |
| Office2003Olive.xaml | Themes | Office 2007 Olive テーマのテンプレートを指定します。 |
| Office2003Royale.xaml | Themes | Office 2007 Royal テーマのテンプレートを指定します。 |
| Office2003Silver.xaml | Themes | Office 2007 Silver テーマのテンプレートを指定します。 |
| Office2007Black.xaml | Themes | Office 2007 Black テーマのテンプレートを指定します。 |
| Office2007Blue.xaml | Themes | Office 2007 Blue テーマのテンプレートを指定します。 |
| Office2007Silver.xaml | Themes | Office 2007 Silver テーマのテンプレートを指定します。 |
| Vista.xaml | Themes | Vista テーマのテンプレートを指定します。 |

時系列グラフ

時系列グラフは、X 軸に時間を表示します。これは非常に一般的なグラフであり、時間の経過に伴う値の変化を示すために使用されます。

ほとんどの時系列グラフでは、時間間隔は一定です(年、月、週、日)。この場合、時系列グラフは、上で説明したような、単純な値タイプのグラフと本質的に同じです。違いは、時系列グラフでは X 軸にカテゴリではなく日付や時刻が表示されることです(時間間隔が一定でない場合、そのグラフは次のセクションで説明する XY グラフになります)。

次に、いくつかの時系列グラフの作成手順を説明します。

手順1: グラフタイプの選択

このコードは、既存の系列をすべてクリアして、グラフタイプを設定します。

```
C#  
  
public Window1 ()  
{  
    InitializeComponent();  
    // 現在のグラフをクリア  
    c1Chart.Reset(true);  
    // グラフのタイプを設定  
    c1Chart.ChartType = ChartType.Column;  
}
```

手順2: 軸の設定

前のサンプルと同様に、まず両方の軸への参照を取得します。**横棒グラフ**のタイプでは軸が逆転することを思い出してください(値は Y 軸に表示されます)。

```
C#  
  
// 軸を取得  
Axis valueAxis = c1Chart.View.AxisY;  
Axis labelAxis = c1Chart.View.AxisX;  
if (c1Chart.ChartType == ChartType.Bar)
```

```

{
    valueAxis = _clChart.View.AxisX;
    labelAxis = _clChart.View.AxisY;
}

```

次に、タイトルを軸に割り当てます。軸のタイトルは単なるテキストではなく、**UIElement** オブジェクトです。ここでは、以前行ったのと同様に、**CreateTextBlock** メソッドを使用して軸のタイトルを設定します。注釈の書式、最小値、および主単位も設定します。違いは、各値の間の目盛記号の間隔が大きくなることです。

C#

// ラベルの軸を設定

```

labelAxis.Title = CreateTextBlock("日付", 14, FontWeights.Bold);
labelAxis.AnnoFormat = "MMM-yy";

```

// 値の軸を設定

```

valueAxis.Title = CreateTextBlock("金額 (万円) ", 14, FontWeights.Bold);
valueAxis.AnnoFormat = "#,##0 ";
valueAxis.MajorUnit = 1000;
valueAxis.AutoMin = false;
valueAxis.Min = 0;

```

手順3:1つ以上のデータ系列の追加

今回は、前に定義した2つめのデータ提供メソッドを使用します。

C#

// データを取得

```

var data = GetSalesPerMonthData();

```

次に、ラベルの軸に日付を表示します。これを行うには、データレコードの各 **Date** 値を取得する Linq ステートメントを使用します。結果は配列に変換され、ラベルの軸の **ItemsSource** プロパティに割り当てられます。

C#

```

clChart.Data.ItemNames = (from r in data select r.Date.ToString("MMM-yy")).Distinct().ToArray();

```

Distinct Linq 演算子を使用して重複した日付の値を削除したことに注意してください。これが必要なのは、今回のデータに、日付ごとに製品あたり1つのレコードが含まれるためです。

これで、グラフに追加される実際の **DataSeries** オブジェクトを作成する準備ができました。各系列は、特定の製品の収益を示します。これは、下記の Linq ステートメントで行うことができます。この例は以前使用したものよりやや複雑ですが、Linq で実現される機能の優れた実用例といえます。

C#

// 製品あたり1つの系列(収益)を追加

```

var products = (from p in data select p.Product).Distinct();
foreach (string product in products)
{
    var ds = new DataSeries();
    ds.Label = product;
    ds.ValuesSource = (
        from r in data

```

Chart for WPF/Silverlight

```
where r.Product == product
select r.Revenue).ToArray();
clChart.Data.Children.Add(ds);
}
```

コードでは、まずデータソース内の製品のリストを作成しています。次に、製品ごとに1つの **DataSeries** を作成します。データ系列のラベルは、単純に製品名です。実際のデータは、現在の製品に属するレコードを抽出して、それらの **Revenue** プロパティを取得することによって得られます。結果は以前と同様、データ系列の **ValuesSource** プロパティに割り当てられます。

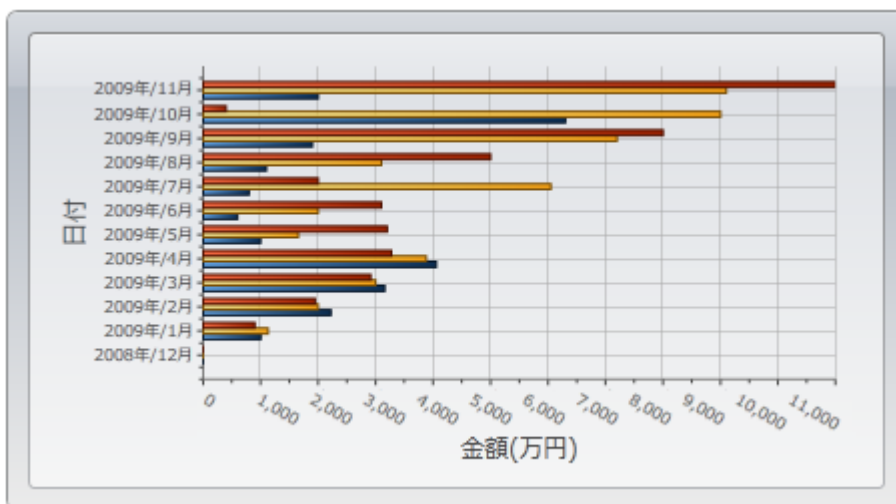
手順4: グラフの外観の調整


今回も、最後に Theme プロパティと Palette プロパティを設定して、グラフの外観を手早く設定します。

```
C#
clChart.Theme = clChart.TryFindResource(
    new ComponentResourceKey(typeof(Cl.WPF.ClChart.ClChart),
        "Office2007Black")) as ResourceDictionary;
```

これで、時系列グラフを生成するコードは終わりです。結果は、以下の図のようになるはずですが。

ChartType.Bar



 **注意:** 上の図では、X 軸と Y 軸のラベルの余地を生み出すため、AnnoAngle プロパティが「30」に設定されています。

各月の第1日におけるデータラベルの表示

各月の第1日にのみデータラベルを表示するには、次のコードを使用します。

VisualBasic

```
clChart1.View.AxisX.IsTime = True
clChart1.View.AxisX.AnnoFormat = "MMM-dd"
` 時間軸で MajorUnit=31 のとき、グラフでは、月内の日数が変化することを考慮して、各月の第1日をマーキングすることが必要
clChart1.View.AxisX.MajorUnit = 31
```



C#

```

c1Chart1.View.AxisX.IsTime = true;
c1Chart1.View.AxisX.AnnoFormat = "MMM-dd";
// 時間軸で MajorUnit=31 のとき、グラフでは、月内の日数が変化することを考慮して、各月の第1日をマーキングすることが必要
c1Chart1.View.AxisX.MajorUnit = 31;

```

傾向線

 **注意:** 傾向線機能を使用するには、C1.WPF.C1Chart.Extended.dll または C1.Silverlight.C1Chart.Extended.dll への参照をプロジェクトに追加する必要があります。


C1Chart では、10 種類の傾向線がサポートされており、傾向線機能を簡単に追加できます。傾向線をチャートに追加する方法と外観を制御するだけです。

C1Chart コントロールでサポートされている傾向線には、回帰傾向線と非回帰傾向線の2つのグループがあります。


平均、最大、最小、および移動平均傾向線は、非回帰傾向線です。

多項式、指数、対数、累乗、およびフーリエ関数は、これらの関数によって傾向が示されるデータを近似する回帰傾向線です。

FitType プロパティを変更することで、**C1Chart** のデータに合わせて傾向線の種類を変更できます。また、**Order** プロパティを変更すると、傾向線とデータポイントの間の適合度を密にまたは疎に設定できます。

 **注意:** 傾向線は、どの種類の 2D チャートでも使用されるわけではありません。一般には、X-Y 折れ線グラフ、横棒グラフ、または散布図で使用されます。

C1Chart への傾向線の追加

 **注意:** このトピックで紹介するコードは、ブログ投稿「Charting Trendlines (傾向線の描画)」からの抜粋です。ダウンロード可能なサンプルは、ブログ投稿「Charting Trendlines (傾向線の描画)」にあります。

傾向線の追加方法は、第2データ系列の追加とほとんど同じです。C1Chart は、値に基づいて自動的に計算を行い、傾向線をプロットします。デフォルトでは、傾向線の **FitType** は Polynomial (多項式)、**Order** は2です。次のコードサンプルに傾向線の追加方法を示します。

Visual Basic

```

' 傾向線を追加します
Dim tl As New TrendLine()
tl.Label = "Trendline"
tl.ConnectionStroke = New SolidColorBrush(Colors.Red)
tl.XValuesSource = myXValues
tl.ValuesSource = myValues
chart.Data.Children.Add(tl)

```

C#

```

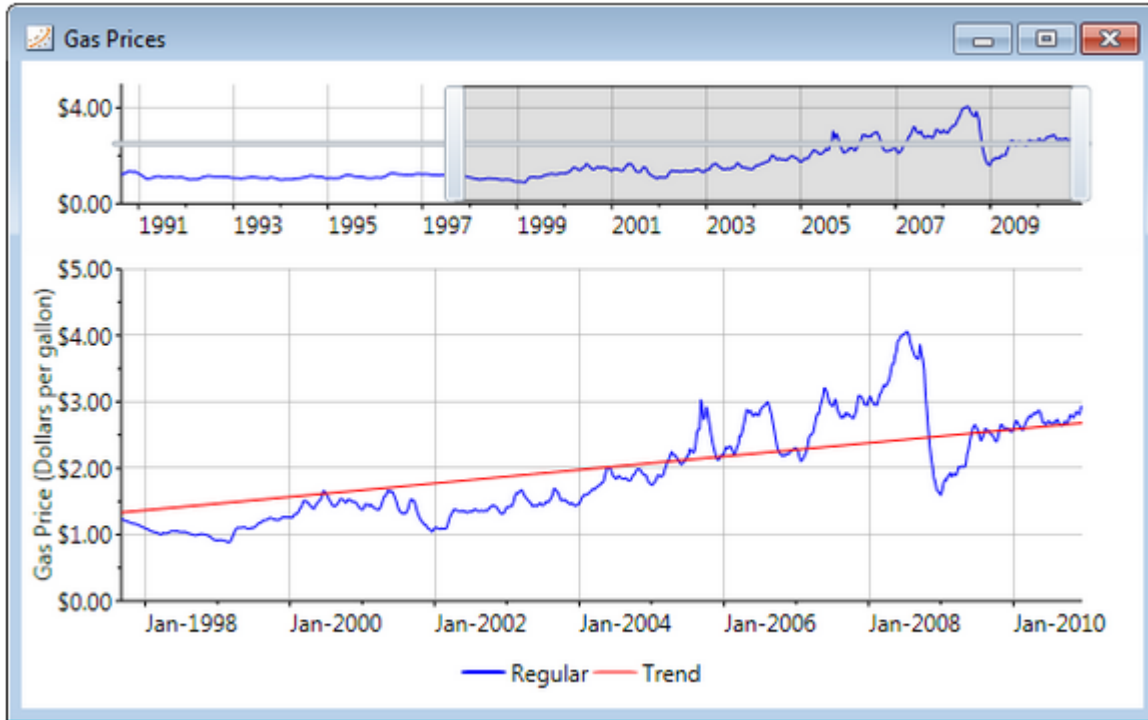
// 傾向線を追加します

```

Chart for WPF/Silverlight

```
TrendLine tl = new TrendLine();  
tl.Label = "Trendline";  
tl.ConnectionStroke = new SolidColorBrush(Colors.Red);  
tl.XValuesSource = myXValues;  
tl.ValuesSource = myValues;  
chart.Data.Children.Add(tl);
```

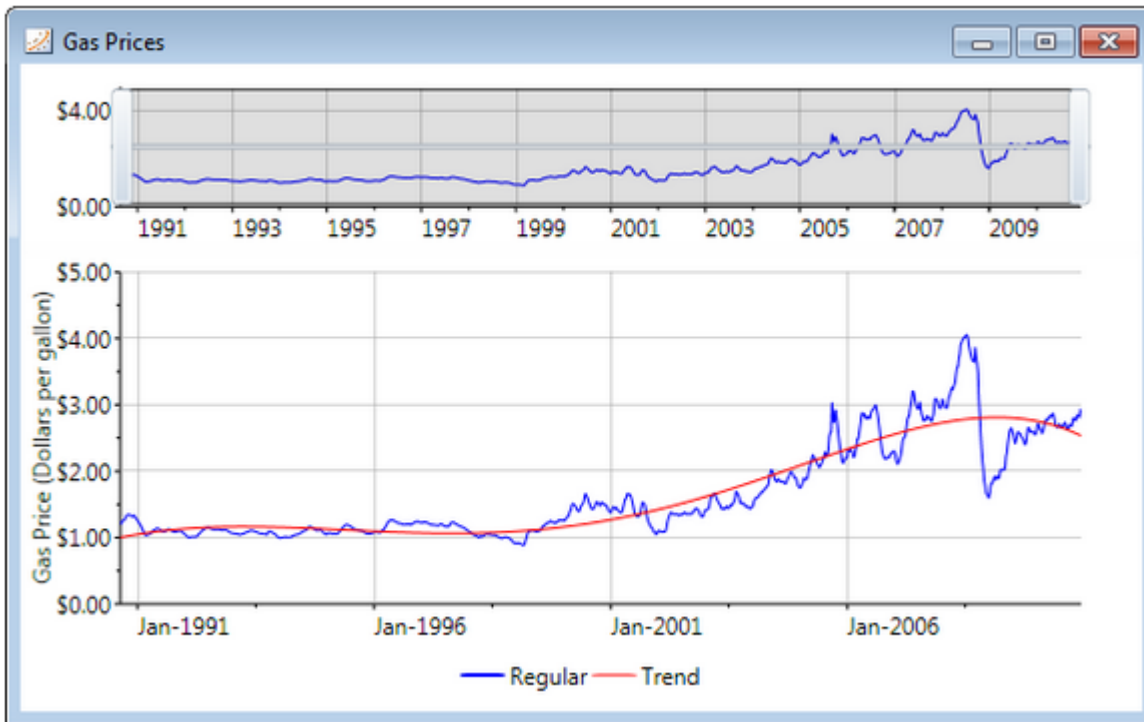
次の図からもわかるように、デフォルトの傾向線は、データに密に適合しません。



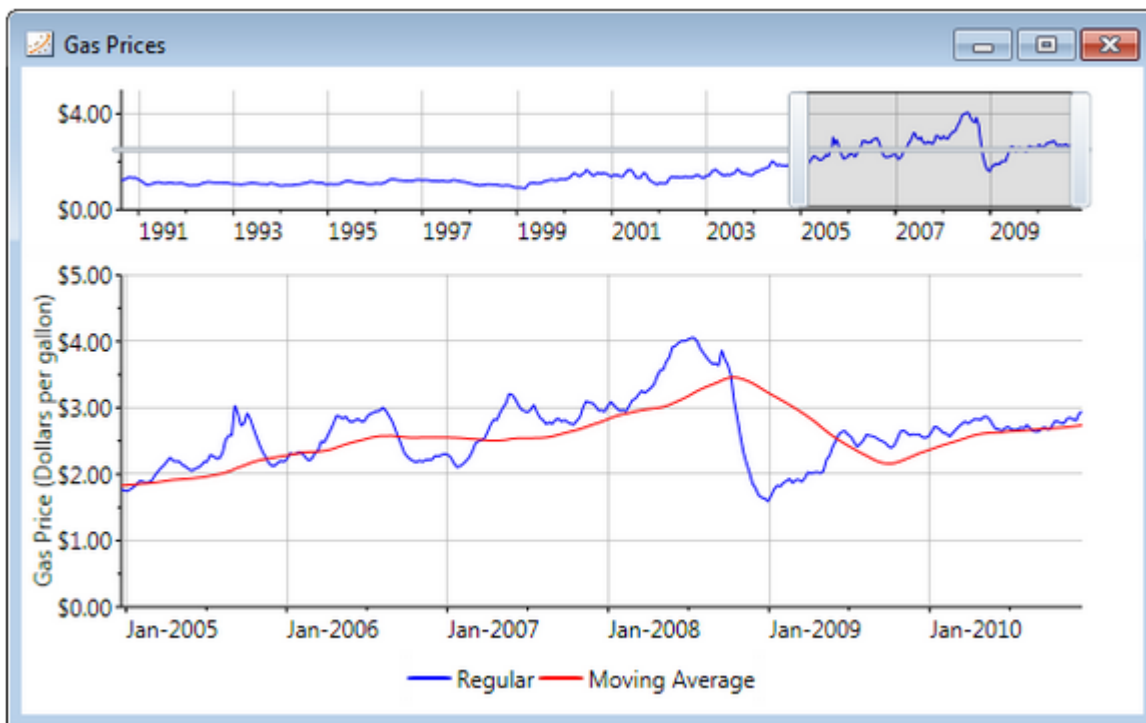
Order プロパティを変更することで、より密に適合させることができます。FitType プロパティと Order プロパティを指定することで、自由に適合度を設定できます。

```
C#  
tl.Label = "Trend";  
tl.FitType = FitType.Polynomial;  
tl.Order = 6;
```

上のコードは、曲線の多項式傾向線を生成しますが、これでもまだ、データに最適に適合しているとはいえません。



図のグラフに使用したデータは、7日ごとのガス価格の値です。この種のデータでは、**移動平均傾向線**がより適しています。移動平均傾向線を使用すると、次の図のように、データによく適合します。



移動平均傾向線を設定する場合は、新しい **MovingAverage** オブジェクトをインスタンス化し、**Period** プロパティを設定する必要があります。このプロパティは、傾向線に使用するデータポイント数を指定します。ガス価格データは7日ごとの平均値なので、**Period** を 48 に設定すると、1年のデータ平均値が計算されます。

C#

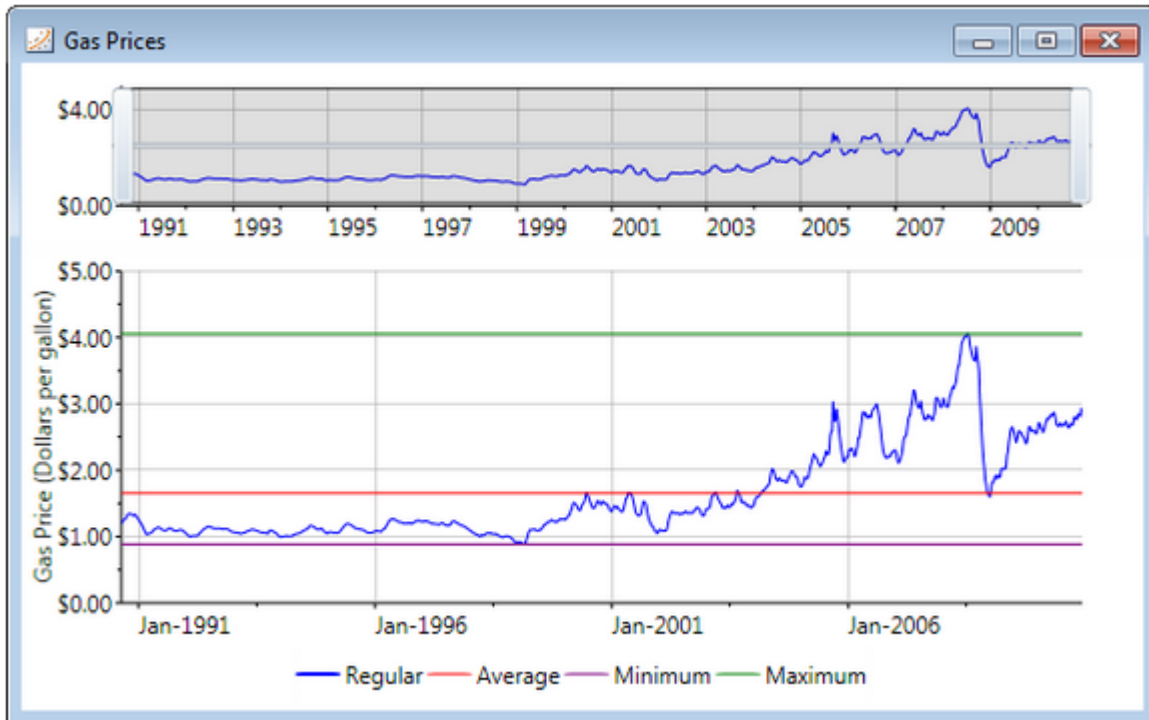
```
MovingAverage ma = new MovingAverage();
ma.Label = "Moving Average";
ma.Period = 48;
```

Chart for WPF/Silverlight

```
ma.XValuesSource = days;  
ma.ValuesSource = price;  
ma.ConnectionStroke = new SolidColorBrush(Colors.Red);  
chart.Data.Children.Add(ma);
```

非回帰傾向線

チャートデータから、最小値、最大値、平均値などの単純なデータに注目する場合は、この3種類の傾向線をインスタンス化し、FitType をそれぞれ MinimumX、MaximumX、AverageX に設定します。




チュートリアル

データバインディングのチュートリアル (WPF のみ)

以下のセクションには、**C1Chart** コントロールのデータバインディングのチュートリアルが含まれています。これらのチュートリアルでは、手順を追って説明します。この章で概説されている手順を実行すれば、**C1Chart** をデータテーブルや XML ファイルにバインドできるようになります。

両方のチュートリアルで使用する主なプロパティは、次のとおりです。

- **ItemsSource**: オブジェクトのリストを提供します。
- **ItemNameBinding**: 要素 (x 軸のラベル) の名前を指定します。
- **ValueBinding**: 数値を指定します。

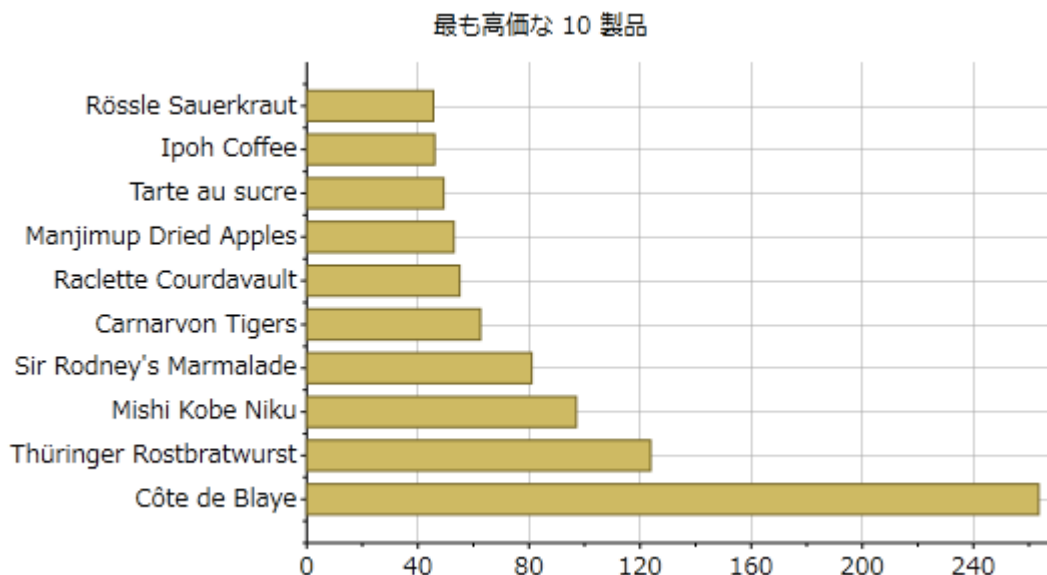
 **メモ**: このセクションの内容は、ComponentOne Studio for WPF にのみ適用されます。

プログラムによるデータテーブルへのバインド

このチュートリアルでは、**C1Chart** をプログラムでデータセットにバインドするための手順を説明します。データは、製品の名前を表す 1 本の y 軸と、各製品の単価を表す 1 本の x 軸を持つ単純な横棒グラフとして情報を表示します。最も高価な 10 種類の製品が降順で表示されます。横棒グラフでは、1 つの系列を使用して単価を表示します。系列が 1 つしかないため、凡例は使用しません。

このグラフは、サンプルの **Access** データベース **Nwind.mdb** のデータを使用します。このクイックスタートでは、データベースファイル **Nwind.mdb** が **c:\Program Files\Common Files\C1Studio\Data** ディレクトリに置かれているものとし、このデータベースに言及する際は、簡潔にするため、フルパスではなくファイル名を使用します。

このチュートリアルを完了すると、グラフの表示は次のようになります。



C1Chart をプログラムでデータテーブルにバインドするには、以下の手順を実行します。

1. Visual Studio で新しい WPF プロジェクトを作成します。
2. **C1.WPF.C1Chart** の参照をプロジェクトに追加します。
3. **C1Chart** コントロールをウィンドウに追加します。

Chart for WPF/Silverlight

4. **C1Chart** コントロールがウィンドウに配置されたら、次の XAML コードを追加します。

XAML

```
Title="Window1" Height="300" Width="500" xmlns:c1chart="clr-namespace:C1.WPF.C1Chart;assembly=C1.WPF.C1Chart" Loaded="Window_Loaded">
  <Grid>
    <c1chart:C1Chart Content="" Margin="10,10,10,18" Name="c1Chart1">
      <c1chart:C1Chart.Data>
        <c1chart:ChartData>
          <c1chart:ChartData.ItemNames>P1 P2 P3 P4
P5</c1chart:ChartData.ItemNames>
          <c1chart:DataSeries Label="系列1" Values="20 22 19 24 25" />
          <c1chart:DataSeries Label="系列2" Values="8 12 10 12 15" />
        </c1chart:ChartData>
      </c1chart:C1Chart.Data>
      <c1chart:Legend DockPanel.Dock="Right" />
    </c1chart:C1Chart>
  </Grid>
```

5. 「XAML」タブを選択して、次の名前空間を XAML コードに追加します。

XAML

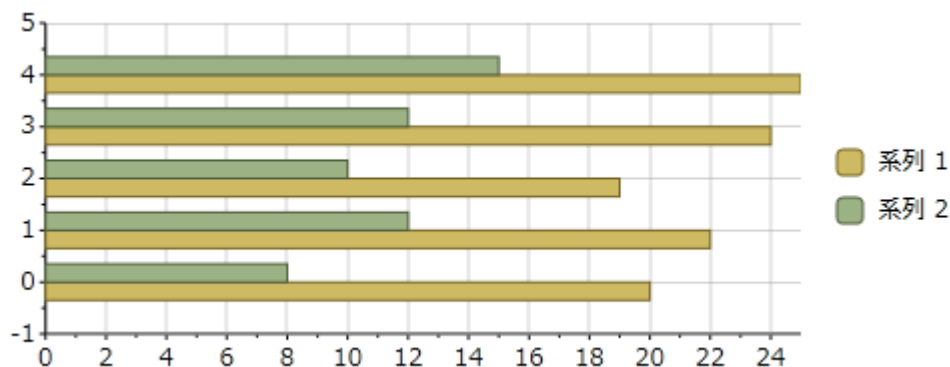
```
xmlns:sys="clr-namespace:System;assembly=mscorlib"
```

6. XAML コードで、タイトルの **Width** を「300」から「500」に変更します。
7. <c1chart:C1Chart> タグ内で、**Margin** を0に変更して、**ChartType** を **Bar** に設定します。これによって、デフォルトのグラフの外観が縦棒から横棒に変わります。XAML コードは次のようになるはずですが。

XAML

```
<c1chart:C1Chart Margin="0" Name="c1Chart1" ChartType="Bar">
</c1chart:C1Chart>
```

グラフは次のように表示されます。



8. c1chart:C1Chart の終了タグの後にラベルを作成して、そのテキストを「最も高価な 10 製品」とします。

XAML

```
<TextBlock DockPanel.Dock="Top" Text="最も高価な 10 製品"
HorizontalAlignment="Center"/>
```

これで、XAML コードは次のようになるはずです。

```
XAML
<Grid>
    <clchart:C1Chart Margin="0" Name="c1Chart1" ChartType="Bar" Height="185"
    VerticalAlignment="Top">
        <clchart:C1Chart.Data>
            <clchart:ChartData>
                <clchart:DataSeries Label="系列1" Values="20 22 19 24 25" />
                <clchart:DataSeries Label="系列2" Values="8 12 10 12 15" />
            </clchart:ChartData>
        </clchart:C1Chart.Data>
        <clchart:Legend DockPanel.Dock="Right" />
    </clchart:C1Chart>
    <TextBlock DockPanel.Dock="Top" Text="最も高価な 10 製品"
    HorizontalAlignment="Center"/>
</Grid>>
```

9. 次の **using** 指示文または **imports** 指示文を、コードビハインドファイルに追加します。

Visual Basic

```
Imports System.Data
Imports System.Data.OleDb
Imports Cl.WPF.C1Chart
```

C#

```
using System.Data;
using System.Data.OleDb;
using Cl.WPF.C1Chart;
```

10. データセットの変数を **Window_Loaded** プロシージャの外部で宣言して、製品と単価をデータベースから取得するために次のコードを追加します。

Visual Basic


```
Private _dataSet As DataSet
Private Sub Window_Loaded(ByVal sender As Object, ByVal e As RoutedEventArgs)
    ' コレクションを作成して、データセットに入力
    Dim mdbFile As String = "c:\Program Files\Common
Files\C1Studio\Data\nwind.mdb"
    Dim connString As String = String.Format("Provider=Microsoft.Jet.OLEDB.4.0;
Data Source={0}", mdbFile)
    Dim conn As New OleDbConnection(connString)
    Dim adapter As New OleDbDataAdapter("SELECT TOP 10 ProductName, UnitPrice" &
Chr(13) & "" & Chr(10) & " FROM Products ORDER BY UnitPrice DESC;", conn)
    _dataSet = New DataSet()
```

Chart for WPF/Silverlight

```
adapter.Fill(_dataSet, "Products")
' グラフのデータのソースを設定
c1Chart1.Data.ItemsSource = _dataSet.Tables("Products").Rows
End Sub
```

C#

```
DataSet _dataSet;
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    // コレクションを作成して、データセットに入力
    string mdbFile = @"c:\Program Files\Common Files\C1Studio\Data\nwind.mdb";
    string connString = string.Format("Provider=Microsoft.Jet.OLEDB.4.0; Data
Source={0}", mdbFile);
    OleDbConnection conn = new OleDbConnection(connString);
    OleDbDataAdapter adapter =
new OleDbDataAdapter(@"SELECT TOP 10 ProductName, UnitPrice
FROM Products ORDER BY UnitPrice DESC;", conn);
    _dataSet = new DataSet();
    adapter.Fill(_dataSet, "Products");
    // グラフのデータのソースを設定
    c1Chart1.Data.ItemsSource = _dataSet.Tables["Products"].Rows;
}
```

 **注意:** mdbFile のファイルパスは、**nwind.mdb** データベースプロジェクトが置かれているマシン上の位置と一致するようにしてください。

11. 「XAML」タブをクリックして XAML ビューを表示し、次のデフォルトデータを ChartData から削除します。

XAML

```
<c1chart:ChartData.ItemNames>P1 P2 P3 P4 P5</c1chart:ChartData.ItemNames>
    <c1chart:DataSeries Label="系列1" Values="20 22 19 24 25" />
    <c1chart:DataSeries Label="系列2" Values="8 12 10 12 15" />
```

これで、ウィンドウ上の **C1Chart** コントロールの表示は空になります。

12. <c1chart:C1Chart.Data> タグ内で、**ItemNameBinding** プロパティを **ChartData** に追加して要素(この場合は y 軸のラベル)の名前を指定し、**ValueBinding** プロパティを **DataSeries** に追加して系列の数値を指定します。

XAML

```
<c1chart:ChartData ItemNameBinding="{Binding Path=[ProductName]}">
    <c1chart:DataSeries ValueBinding="{Binding Path=
[UnitPrice]}" />
</c1chart:ChartData>
```

プロジェクトの XAML コードは、次のようになるはずですが。

XAML

```
<Window x:Class="Chart for WPF_QuickStart.Window1"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:sys="clr-namespace:System;assembly=microsoftlib"
```



```

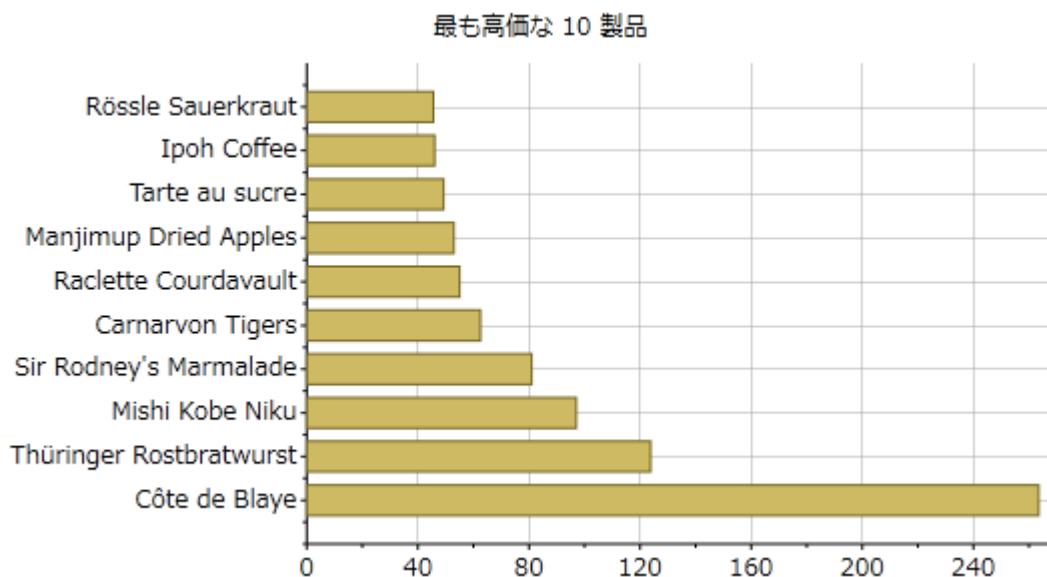
Title="Window1" Height="300" Width="500" Loaded="Window_Loaded"
xmlns:clchart="clr-namespace:C1.WPF.C1Chart;assembly=C1.WPF.C1Chart">
<Grid>
  <clchart:C1Chart Margin="0" Name="c1Chart1" ChartType="Bar">
    <TextBlock DockPanel.Dock="Top" Text="最も高価な 10 製品"
      HorizontalAlignment="Center"/>
    <clchart:C1Chart.Data>
      <clchart:ChartData ItemNameBinding="{Binding Path=
[ProductName]}">
        <clchart:DataSeries ValueBinding="{Binding Path=
[UnitPrice]}"/>
      </clchart:ChartData>
    </clchart:C1Chart.Data>
  </clchart:C1Chart>
</Grid>
</Window>

```

13. <clchart:Legend DockPanel.Dock="Right" /> タグを XAML から削除して、組み込みの Legend コントロールを削除します。
14. プロジェクトを実行して、すべてが正しく機能していることを確認します。

実行時に次のことを確認してください。

グラフに **Products** テーブルのデータが表示されます。



XML へのバインド

このチュートリアルでは、XML を XAML ページのソースにデータアイランドとして埋め込んで、C1Chart コントロールを XML データにバインドするための手順を説明します。データは、都市の名前を表す1本の y 軸と、各国の人口を表す1本の x 軸を持つ単純な横棒グラフとして情報を表示します。横棒グラフでは、1つの系列を使用して人口を表示します。人口の色を示すため、凡例を使用します。

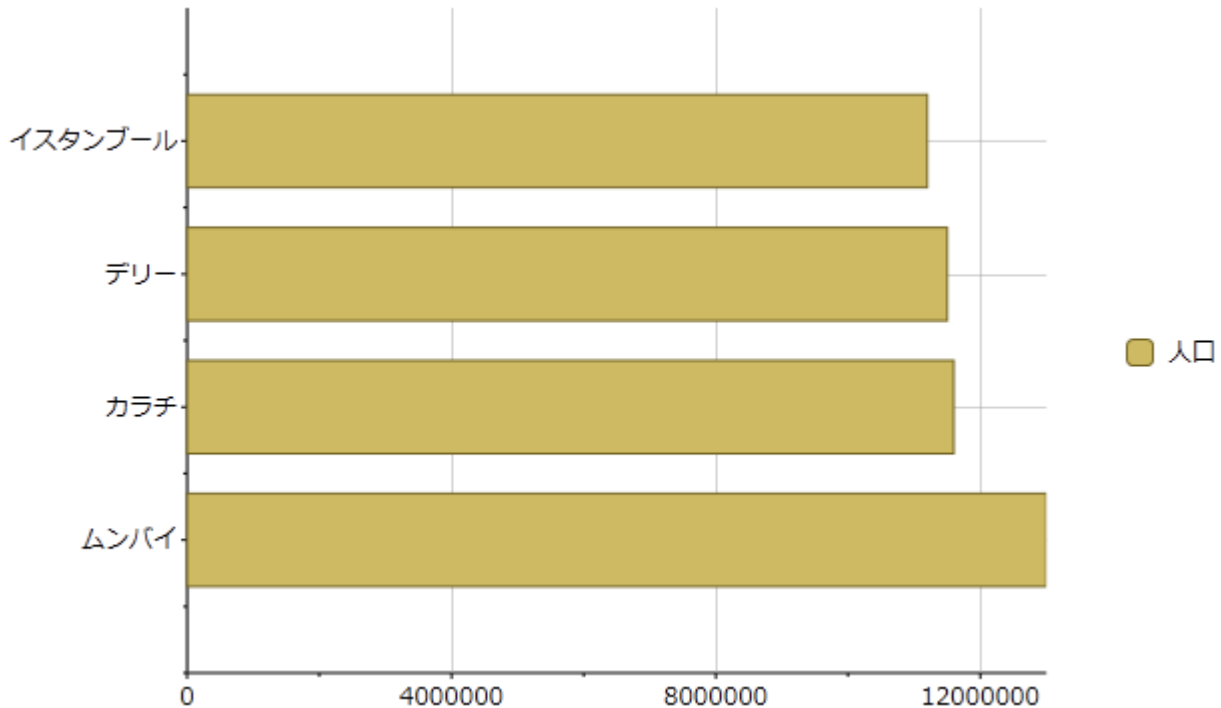
このチュートリアルでは、次の XAML コードを使用して、**ChartData** クラスでバインディングを設定します。

XAML

Chart for WPF/Silverlight

```
<clchart:ChartData ItemsSource="{Binding Source={StaticResource data}}"
  ItemNameBinding="{Binding XPath=CityName}">
  <clchart:DataSeries Label="人口"
    ValueBinding="{Binding XPath=Population}" /> </clchart:ChartData>
```

このチュートリアルを完了すると、グラフの表示は次のようになります。



C1Chart を XML にバインドするには、以下の手順を実行します。

1. Visual Studio で新しい WPF プロジェクトを作成します。
2. ウィンドウでリソースセクションを作成して、そこに XML データプロバイダを追加します。リソースセクション内に、XML データを XML データアイランドとして直接埋め込みます。XML データアイランドは `<x:Xdata>` タグで囲み、ルートノードは1つだけ(この例では Cities)にする必要があります。

XAML

```
<Grid.Resources>
  <XmlDataProvider x:Key="data" XPath="Cities/City">
    <x:XData>
      <Cities xmlns="">
        <City>
          <CityName>ムンバイ</CityName>
          <Population>1300000</Population>
        </City>
        <City>
          <CityName>カラチ</CityName>
          <Population>1160000</Population>
        </City>
        <City>
          <CityName>デリー</CityName>
          <Population>1150000</Population>
        </City>
        <City>
          <CityName>イスタンブール</CityName>
```

```

        <Population>11200000</Population>
    </City>
</Cities>
</x:XData>
</XmlDataProvider>
</Grid.Resources>

```

3. **C1.WPF.C1Chart** の参照をプロジェクトに追加します。
4. **C1Chart** コントロールをウィンドウに追加します。
C1Chart コントロールがウィンドウに配置されたら、次の XAML コードを追加します。

XAML

```

Title="Window1" Height="50" Width="100" xmlns:c1chart="clr-
namespace:C1.WPF.C1Chart;assembly=C1.WPF.C1Chart" Loaded="Window_Loaded">
    <Grid>
        <c1chart:C1Chart Content="" Margin="10,10,10,18" Name="c1Chart1">
            <c1chart:C1Chart.Data>
                <c1chart:ChartData>
                    <c1chart:ChartData.ItemNames>P1 P2 P3 P4
P5</c1chart:ChartData.ItemNames>
                    <c1chart:DataSeries Label="系列1" Values="20 22 19 24 25" />
                    <c1chart:DataSeries Label="系列2" Values="8 12 10 12 15" />
                </c1chart:ChartData>
            </c1chart:C1Chart.Data>
            <c1chart:Legend DockPanel.Dock="Right" />
        </c1chart:C1Chart>
    </Grid>

```

5. ウィンドウの **Width** を 300 に、高さを 550 に変更します。
6. タグ内で、**Margin** を 0 に変更して、**ChartType** を **Bar** に設定します。これによって、デフォルトのグラフの外観が縦棒から横棒に変わります。XAML コードは次のようになるはずですが。

XAML

```

<c1chart:C1Chart Margin="0" Name="c1Chart1" ChartType="Bar">
</c1chart:C1Chart>

```

7. XAML ファイルで、<c1chart:C1Chart.Data> タグを見つけて、次の XAML コードを削除します。

XAML

```

<c1chart:ChartData.ItemNames>P1 P2 P3 P4 P5</c1chart:ChartData.ItemNames>
    <c1chart:DataSeries Label="系列1" Values="20 22 19 24 25" />
    <c1chart:DataSeries Label="系列2" Values="8 12 10 12 15" />

```

2つのデフォルト系列が C1Chart から削除され、データがないために C1Chart コントロールの表示は空になります。

8. <c1chart:C1Chart.Data> タグ内で、**ItemNameBinding** プロパティを **ChartData** に追加して要素(この場合は y 軸のラベル)の名前を指定し、**ValueBinding** プロパティを **DataSeries** に追加して系列の数値を指定します。次の例は、ソースを指定するバインディングエクステンションを使用して、**ChartData.ItemsSource** プロパティをバインドしています。**ChartData.ItemNameBinding** プロパティは、パスを指定するバインディングエクステンションを使用してバインドされます。**DataSeries.Label** プロパティは、パス (Population) を指定するバインディングエクステンションを使用してバインドされます。

XAML の場合:

XAML

Chart for WPF/Silverlight

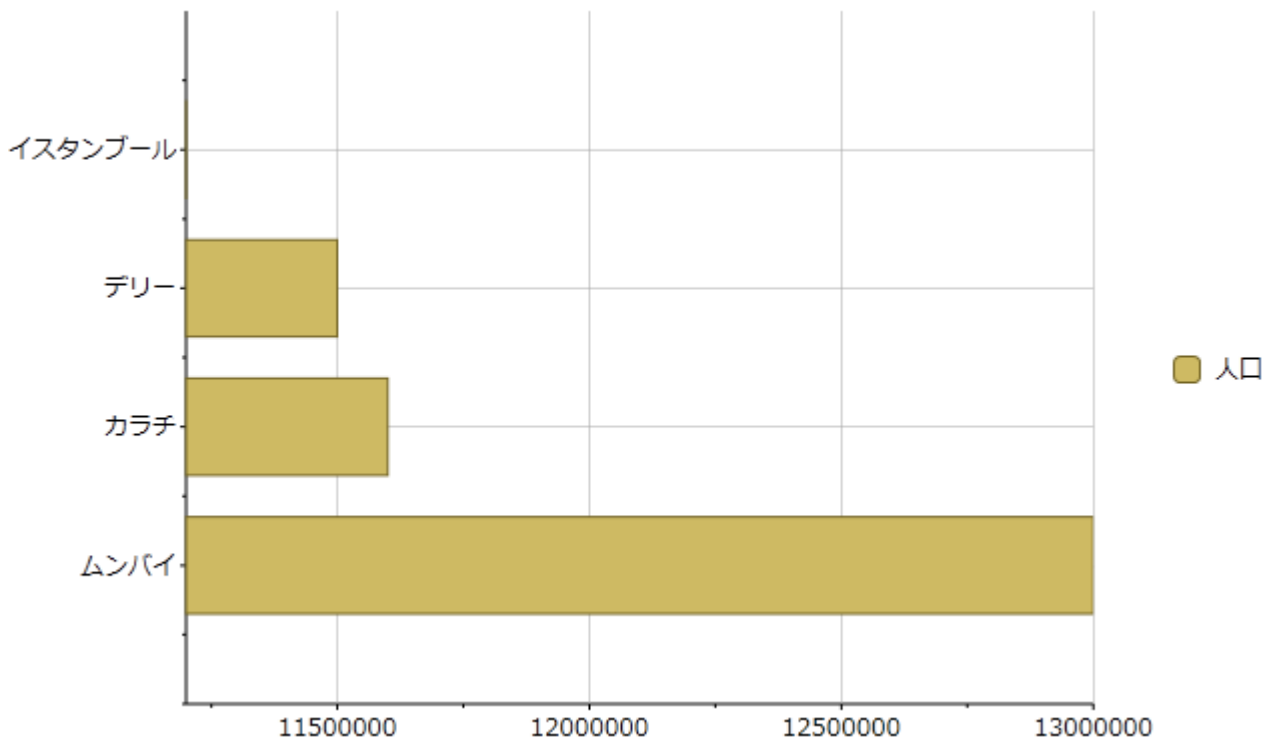
```
<clchart:ChartData ItemsSource="{Binding Source={StaticResource data}}"
    ItemNameBinding="{Binding XPath=CityName}">
    <clchart:DataSeries Label="人口"
        ValueBinding="{Binding XPath=Population}" />
</clchart:ChartData>
```

C1Chart コントロールの XAML コードは、次のようになるはずですが。

XAML

```
<clchart:C1Chart Height="300" HorizontalAlignment="Left" Margin="0"
Name="c1Chart1" ChartType="Bar" VerticalAlignment="Top" Width="500">
    <clchart:C1Chart.Data>
        <clchart:ChartData ItemsSource="{Binding Source={StaticResource
data}}"
            ItemNameBinding="{Binding XPath=CityName}">
            <clchart:DataSeries Label="人口"
                ValueBinding="{Binding XPath=Population}" />
        </clchart:ChartData>
    </clchart:C1Chart.Data>
    <clchart:Legend DockPanel.Dock="Right" />
</clchart:C1Chart>
```

9. プロジェクトを実行して、すべてが正しく機能していることを確認します。グラフは次のように表示されます。



x 軸の注釈の表示に注意してください。x 軸の注釈を書式設定して、人口が桁区切り付きで表示される必要があります。

10. **C1Chart** の **ChartView.AxisX** プロパティのタグを宣言します。以下のように **AxisX** のプロパティを設定して、注釈とグリッド線を書式設定する必要があります。終了タグ `</c1chart:C1Chart.Data>` の後に、次の XAML コードを追加します。

XAML

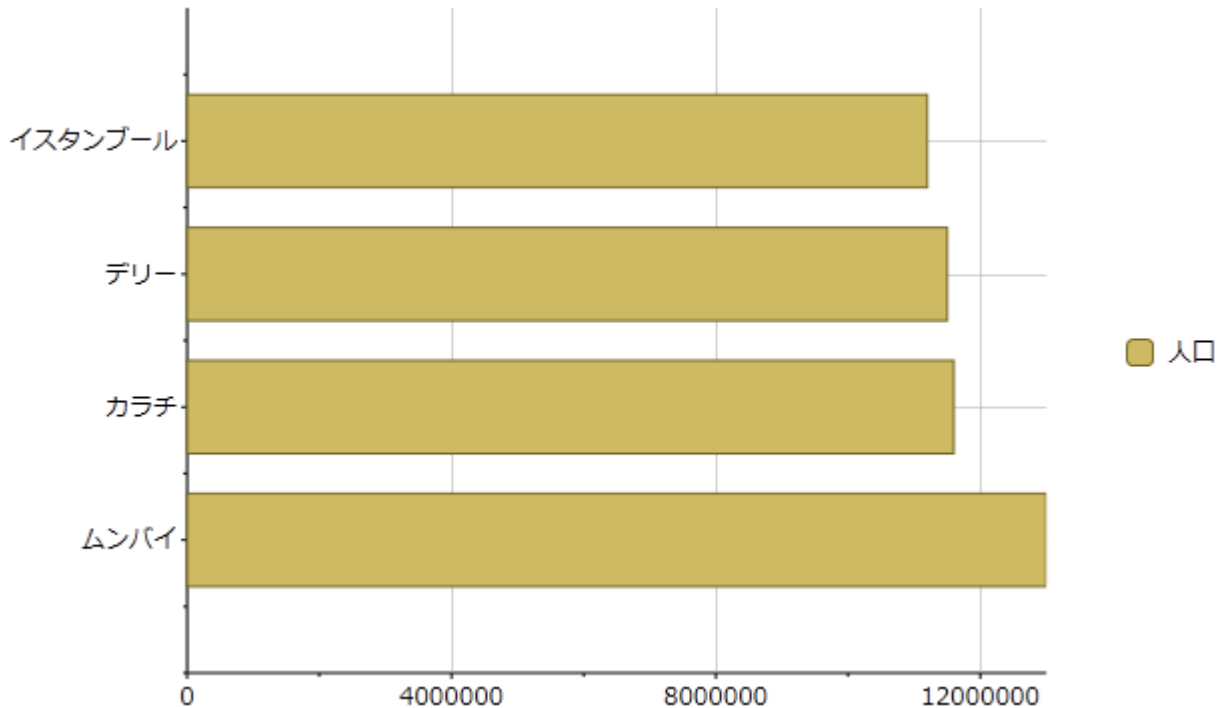
```
<clchart:C1Chart.View>
```

```

<clchart:ChartView>
  <clchart:ChartView.AxisX >
    <clchart:Axis Min="0" MajorGridStroke="DarkGray"
AnnoFormat="#,###,###"/>
  </clchart:ChartView.AxisX>
</clchart:ChartView>
</clchart:C1Chart.View>

```

X 軸の注釈は、次のように更新されてグラフ上に表示されます。



MVVM の使用

Chart for WPF/Silverlight は、MVVM (Model-View-ViewModel) デザインパターンをサポートします。WPF ネイティブの連結手法を使用して、XAML でグラフ全体を宣言的に記述して連結することができます。

次の手順では、MVVM 設計のアプリケーションで **C1Chart** を使用する方法を示します。

手順 1: モデルの作成

INotifyPropertyChanged インターフェイスを実装する新しいクラス **Sale** を作成します。

```

C#
public class Sale : INotifyPropertyChanged
{
    private string _product;
    private double _value;
    private double _discount;

    public Sale(string product, double value, double discount)
    {
        Product = product;
    }
}

```

Chart for WPF/Silverlight

```
        Value = value;
        Discount = discount;
    }

    public string Product
    {
        get { return _product; }
        set
        {
            if (_product != value)
            {
                _product = value;
                OnPropertyChanged("Product");
            }
        }
    }

    public double Value
    {
        get { return _value; }
        set
        {
            if (_value != value)
            {
                _value = value;
                OnPropertyChanged("Value");
            }
        }
    }

    public double Discount
    {
        get { return _discount; }
        set
        {
            if (_discount != value)
            {
                _discount = value;
                OnPropertyChanged("Discount");
            }
        }
    }

    public event PropertyChangedEventHandler PropertyChanged;

    void OnPropertyChanged(string propertyName)
    {
        if (PropertyChanged != null)
            PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
    }
}
```

このクラスは、Sale を定義するプロパティとして、Product、Value、および Discount を持ちます。

INotifyPropertyChanged を実装することで、連結プロパティが自動的に動的変更を反映するようになります。変更通知が必要な各プロパティに対しては、プロパティが更新されるたびに **OnPropertyChanged** を呼び出します。ObservableCollections は、既に **INotifyPropertyChanged** を継承していることに注意してください。

手順 2: ビューモデルの作成

SaleViewModel という名前の新しいクラスを作成します。これは、**C1Chart** が表示されるビューの DataContext になります。

```
C#
public class SaleViewModel : INotifyPropertyChanged
{
    private ObservableCollection<Sale> _sales = new ObservableCollection<Sale>();

    public SaleViewModel()
    {
        //データを読み込みます
        LoadData();
    }

    public ObservableCollection<Sale> Sales
    {
        get { return _sales; }
    }

    public void LoadData()
    {
        //TODO: データソースからデータを読み込みます
        _sales.Add(new Sale("Bikes", 23812.89, 12479.44));
        _sales.Add(new Sale("Shirts", 79752.21, 19856.86));
        _sales.Add(new Sale("Helmets", 63792.05, 16402.94));
        _sales.Add(new Sale("Pads", 30027.79, 10495.43));
    }

    public event PropertyChangedEventHandler PropertyChanged;

    void OnPropertyChanged(string propertyName)
    {
        if (PropertyChanged != null)
            PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
    }
}
```

このクラスには、**ObservableCollection**、**Sales**、および初期化時に模擬データを生成するためのメソッドが含まれます。

手順 3: C1Chart を使用したビューの作成

1. **SaleView.xaml** という名前の新しい WPF/Silverlight UserControl を作成し、LayoutRoot グリッドの上に次の XAML を追加します。

Chart for WPF/Silverlight


```
<UserControl.Resources>
    <local:SaleViewModel x:Key="viewModel" />
</UserControl.Resources>
<UserControl.DataContext>
    <Binding Source="{StaticResource viewModel}"/>
</UserControl.DataContext>
```

この XAML は、SaleViewModel を Resource として宣言し、これを UserControl の DataContext に設定します。これで、実行時にビューが ViewModel に連結されます。また、ビュー内のコントロールを ViewModel のパブリックプロパティに連結できるようになりました。

2. **C1Chart** コントロールをページに追加します。
3. **C1Chart** の XAML を次のコードに置き換えます。

```
<c1:C1Chart ChartType="Column" Name="c1Chart1" Palette="Module">
    <c1:C1Chart.Data>
        <c1:ChartData ItemsSource="{Binding Sales}" ItemNameBinding="{Binding Product}">
            <c1:DataSeries Label="値段" ValueBinding="{Binding Value}" />
            <c1:DataSeries Label="割引" ValueBinding="{Binding Discount}" />
        </c1:ChartData>
    </c1:C1Chart.Data>
    <c1:C1ChartLegend />
</c1:C1Chart>
```

この XAML は、2つのデータ系列を含む **C1Chart** を定義します。ChartData の **ItemsSource** は、ViewModel によって公開される Sales オブジェクトのコレクションに設定されます。各 DataSeries は ValueBinding プロパティが設定されます。また、X 軸に沿って製品名が表示されるように ItemNameBinding も設定されます。

 **メモ:** XYDataSeries を使用する場合は、系列ごとに XValueBinding を指定する必要があります。また、ItemNameBinding は設定すべきではありません。

4. **App.xaml.cs** アプリケーション設定ファイルを開き、**Application_Startup** イベントのコードを次のコードに置き換えます。

```
private void Application_Startup(object sender, StartupEventArgs e)
{
    this.RootVisual = new SaleView();
}
```

このコードは、起動時に SaleView が表示されるように RootVisual を設定します。

5. アプリケーションを実行し、**C1Chart** が ViewModel の Sales データに連結されていることを確認します。

