|  |  |
| --- | --- |
| **Navigation:**  [NinjaScript](https://ninjatrader.com/es/support/helpGuides/nt8/ninjascript.htm) > [Language Reference](https://ninjatrader.com/es/support/helpGuides/nt8/language_reference_wip.htm) > [Indicator](https://ninjatrader.com/es/support/helpGuides/nt8/indicator.htm) >  **IndicatorBaseConverter Class** | [Previous page](https://ninjatrader.com/es/support/helpGuides/nt8/drawverticalgridlines.htm) [Return to chapter overview](https://ninjatrader.com/es/support/helpGuides/nt8/indicator.htm) [Next page](https://ninjatrader.com/es/support/helpGuides/nt8/ischartonly.htm) |

**Definition**

A custom [TypeConverter](https://msdn.microsoft.com/en-us/library/system.componentmodel.typeconverter%28v=vs.110%29.aspx) class handling the designed behavior of an indicator's property descriptor collection.  Use this as a base class for any custom **TypeConverter** you are applying to an indicator class.

|  |
| --- |
| **Notes:**  •A working NinjaScript demo can be found through the reference sample on "[Using a TypeConverter to Customize Property Grid Behavior](http://ninjatrader.com/support/forum/showthread.php?t=97919)"  •When applying the custom converter, you must fully qualify the name (e.g., "NinjaTrader.NinjaScript.Indicators.MyCustomConveter")  •Additional **TypeConverter** information can be found from the [MSDN documentation](https://msdn.microsoft.com/en-us/library/system.componentmodel.typeconverter%28v=vs.110%29.aspx)  •See also [TypeConverterAttribute](https://ninjatrader.com/es/support/helpGuides/nt8/typeconverterattribute.htm)  •For Strategies, see the [StrategyBaseConverter](https://ninjatrader.com/es/support/helpGuides/nt8/strategybaseconverter.htm) class |

**Relevant base methods**

|  |  |
| --- | --- |
| [TypeConverter.GetProperties()](https://msdn.microsoft.com/en-us/library/system.componentmodel.typeconverter.getproperties(v=vs.110).aspx) | When overriding **GetProperties()**, calling base.GetProperties() ensures that all default property grid behavior works as designed |
| [TypeConverter.GetPropertiesSupported()](https://msdn.microsoft.com/en-us/library/system.componentmodel.typeconverter.getpropertiessupported(v=vs.110).aspx) | In your custom converter class, you must override **GetPropertiesSupported()**and return a value of **true** in order for your custom type converter to work |

**Syntax**

public class IndicatorBaseConverter : TypeConverter

|  |
| --- |
| **Warning**:  Failure to apply a type of **IndicatorBaseConverter** on an indicator class can result in unpredictable behavior of the standard NinjaTrader WPF property grid. |

|  |
| --- |
| **Tip**: Common indicator functions like Print() are not available to a type converter instance.  To debug a type converter class, you can use the AddOn [Debug Concepts](https://ninjatrader.com/es/support/helpGuides/nt8/alert_and_debug_concepts.htm) or [attach to a debugger](https://ninjatrader.com/es/support/helpGuides/nt8/visual_studio_debugging.htm) (recommended) |

**Examples**

| ns | |
| --- | --- |
| //This namespace holds Indicators in this folder and is required. Do not change it. namespace NinjaTrader.NinjaScript.Indicators {   // When applying the type converter, you must fully qualify the name   [TypeConverter("NinjaTrader.NinjaScript.Indicators.MyCustomConveter")]   public class MyCustomIndicator : Indicator   {     protected override void OnStateChange()     {         if (State == State.SetDefaults)         {           Name   = "MyCustomIndicator";         }     }       protected override void OnBarUpdate()     {         //Add your custom indicator logic here.     }   }     public class MyCustomConveter : IndicatorBaseConverter   {     // A general TypeConveter method used for converting types     public override PropertyDescriptorCollection GetProperties(ITypeDescriptorContext context, object component, Attribute[] attrs)     {         // sometimes you may need the indicator instance which actually exists on the grid         MyCustomIndicator indicator = component as MyCustomIndicator;           // base.GetProperties ensures we have all the properties (and associated property grid editors)         // NinjaTrader internal logic handles for a given indicator         PropertyDescriptorCollection propertyDescriptorCollection = base.GetPropertiesSupported(context)                 ? base.GetProperties(context, component, attrs) : TypeDescriptor.GetProperties(component, attrs);           if (indicator == null || propertyDescriptorCollection == null)           return propertyDescriptorCollection;           // example of why you may need the instance that exists on the grid....         if (indicator.EntryHandling == EntryHandling.UniqueEntries)         {           // do something in the event a property contains some value...         }           // Loop all of the properties of the indicator         foreach (PropertyDescriptor property in propertyDescriptorCollection)         {           // do something with a specific property             // cannot call Print() here           // but you can call the static Output window "Process()"           NinjaTrader.Code.Output.Process(property.Name, PrintTo.OutputTab1);         }           // must return the collection after making changes         return propertyDescriptorCollection;     }       // Important:  This must return true otherwise the type converter will not be called     public override bool GetPropertiesSupported(ITypeDescriptorContext context)     { return true; }     }   } } | |
| **Navigation:**  [NinjaScript](https://ninjatrader.com/es/support/helpGuides/nt8/ninjascript.htm) >  **Code Breaking Changes** | | [Previous page](https://ninjatrader.com/es/support/helpGuides/nt8/ninjascript.htm) [Return to chapter overview](https://ninjatrader.com/es/support/helpGuides/nt8/ninjascript.htm) [Next page](https://ninjatrader.com/es/support/helpGuides/nt8/ninjascript_best_practices.htm) |

The following document is intended as a high level overview of the NinjaScript changes you can expect between NinjaTrader 7 and NinjaTrader 8.  For specific information on a particular method or property, you can refer to the dynamically formatted **Code Breaking table** at the bottom of this page.  We recommend using the **Filter** and **Sorting** features built into the table, as well checking the **Summary** column and expanding the **Details** section of each entry for general information.  Referring to the conveniently linked NinjaTrader 8 and NinjaTrader 7 documentation will provide specific information on syntax, usage, and examples of any new implementation or element names.

|  |
| --- |
| **Note**:  Information on this page focuses on **supported** **(documented)** NinjaTrader methods and properties shared between versions.  NinjaTrader 8 has seen a significant increase in supported NinjaTrader code, however if you were using previously **undocumented** NinjaTrader 7 methods or properties, they will **NOT** be covered in this topic.  You may be able to find more information on previously **undocumented** methods and properties in the NinjaTrader 8 Help Guide, or our support staff will also be happy to personally point you in the right direction. |

|  |
| --- |
| **Critical**:   If your product uses **unsupported (undocumented)** elements we strongly urge you to put your scripts through thorough testing to ensure they still behave as expected.  There is **NO** guarantee that previously **undocumented** method or property behavior has not changed in the new version of NinjaTrader 8. |

For questions or comments, please contact us at platformsupport@ninjatrader.com

tog_minus        [Implementation Changes Overview](javascript:HMToggle('toggle','ImplementationChangesOverview','ImplementationChangesOverview_ICON'))

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Initialize(), OnStartUp(), OnTermination()**  NinjaTrader 8 has simplified the methods used to set or release various resources during the lifetime of a NinjaTrader object to a single [**OnStateChange()**](https://ninjatrader.com/es/support/helpGuides/nt8/onstatechange.htm) method. This single method is guaranteed to be called for every change in **State** of the object.  It is from this method you can monitor the progression of the object throughout its lifetime in order to setup various resources, set properties, or take action the moment **State** has changed.  This method also exposes a [**State**](https://ninjatrader.com/es/support/helpGuides/nt8/state.htm) variable which can be used in various other methods, such as**OnBarUpdate(),** in order to tell your indicator or strategy to process data depending on the actual **State** of the object.    For example, pushing settings to the UI, or setting initial values for public properties can now be done use **OnStateChange()** when the state has reached**State.SetDefaults**:     | ns | | --- | | protected override void OnStateChange() {   if (State == State.SetDefaults)   {     // set the default properties     Name = "My Indicator";     Fast = 10;     Slow = 25;     IsOverlay = true;     IsAutoScale = true;   } } |       If you have custom resources that need to be setup before the NinjaTrader object is active and processing data, instead of using the**Initialize()** method, you can now set this up once the **OnStateChange()** method has reached **State.Configure** state:     | ns | | --- | | protected override void OnStateChange() {   if (State == State.Configure)   {     // Add a 5 minute Bars object to the strategy     AddDataSeries(Data.BarsPeriodType.Minute, 5);     // setup a custom data series     spread = new Series<double>(this);     // setup a 20-period EMA indicator     ema = EMA(20);     // add indicator to strategy for visual purposes     AddChartIndicator(ema);     } } |       NinjaTrader 7 had no concept to detect when your NinjaTrader object was transitioning from processing Historical data to processing Real-time data.  Now with NinjaTrader 8, the **OnStateChange()** method provides a **State.Transition** state which will notify you when this change is about to occur.  If your NinjaTrader 7 indicators or strategies were using custom methods to try to detect this transition, your custom methods may be refactored under this new state:     | ns | | --- | | protected override void OnStateChange() {   if (State == State.Transition)   {     Print("We're going to real-time data...");     // setup your real-time data resources here   } } |       When your NinjaTrader object is shutting down, and you need clean up any custom device resources, instead of using **OnTermination()**, you should now clean up these resources once the **OnStateChange()** method has reached the **State.Terminated** state:     | ns | | --- | | protected override void OnStateChange() {   if (State == State.Terminated)   {     // release any device resources     if(myTimer != null)         myTimer = null;   } } |     NinjaTrader previously used a **Historical** bool property to notify when an indicator or strategy bar was being processed historically or real-time.  The NinjaTrader 8 **OnStateChange()** approach has now introduced a class level variable **State** where you can check for **State.Historical** or**State.Realtime**in any of the other event methods which will allow you to take action depending on the desired state:     | ns | | --- | | protected override void OnBarUpdate() {   // only process on real-time data   if (State == State.Historical)     return;     else if (State >= State.Realtime)       // rest of logic } |     **Strategies, Orders, and Accounts**  Low level access has been provided to allow more flexibility with the information pertaining to trade data.    •IOrders, IExecution, and IPosition interfaces have all been replaced directly with the corresponding object  •The signatures of the related NinjaScript events have changed to match the NinjaTrader internal Update events  •Methods now return and update with the object instance generated, instead of the previously used interface     |  | | --- | | **Tip**:  Since NinjaTrader 8 now exposes the direct **Order** object, rather than an **IOrder** interface, it is possible to receive **null object reference errors** if you attempt to access an order object before the entry or exit order method has returned.  To prevent these situations, it is recommended to assign your strategies **Order** variables in the **OnOrderUpdate()** method and match them by their **signal name** (order.Name).  Please see the example beginning on line #22 below for demonstration of assigning order objects to private variables. |      | ns | | --- | | Order myOrder = null;   protected override void OnBarUpdate() {           if (Position.MarketPosition == MarketPosition.Flat && myOrder == null)     EnterLongLimit(Low[0], "Entry");     if (myOrder != null)   {     Print(myOrder.OrderState);           if (myOrder.OrderState == OrderState.Cancelled || myOrder.OrderState == OrderState.Filled)         myOrder = null;               } }         protected override void OnOrderUpdate(Cbi.Order order, double limitPrice, double stopPrice,   int quantity, int filled, double averageFillPrice,   Cbi.OrderState orderState, DateTime time, Cbi.ErrorCode error, string comment) {         // compare the order object created via EnterLongLimit by the signal name   if (myOrder == null && order.Name == "Entry")   {     // assign myOrder to matching order update     myOrder = order;           } } |     **Data Series**  Previously there had been type specific Data Series implementations (e.g., IntSeries, TimeSeries, BoolSeries, etc).  Now there just is a template [Series<T>](https://ninjatrader.com/es/support/helpGuides/nt8/seriest.htm) class which could be used generically and even allows support for additional types:     | ns | | --- | | Series<double> mySeries = new Series<double>(this); Series<DateTime> myTimeSeries = new Series<DateTime>(this); |     The **DataSeries.Set()** method used to assign Data Series or Plot values has been removed and values can now be stored using a single assignment operator:     | ns | | --- | | protected override void OnBarUpdate() {   // set public plotting data series close value of current bar   MyPlot[0] = Close[0];   // set custom Series<DateTime> to time value of current bar   myTimeSeries[0] = Time[0];         } |     **Drawing**  The DrawObjects used in NinjaTrader have received a number of changes:    •All DrawObjects have been moved to a separate **NinjaScript.DrawingTools** namespace and are properly known as **DrawingTools**  •Drawing Methods called from indicators or strategies have been moved to a new static partial **Draw** class  •Drawing Methods have all received a signature change which requires you specify the owner (object) which drew the **DrawingTool** object  •Drawing Methods no longer returns an interface but rather an instance of the **DrawingTool** object itself  •Drawing Methods now use the [System.Windows.Media.Brushes](https://msdn.microsoft.com/en-us/library/system.windows.media.brushes%28v=vs.110%29.aspx) class instead of the [System.Drawing.Color](https://msdn.microsoft.com/en-us/library/system.drawing.color(v=vs.110).aspx) structure     |  | | --- | | **Tip**:  DrawingTools are now completely unprotected and you can review their source code from the DrawingTools folder of the NinjaScript Editor's explorer menu |      | ns | | --- | | // example syntax Draw.Line(NinjaScriptBase owner, string tag, int startBarsAgo, double startY, int endBarsAgo, double endY, Brush brush)   // example usage Draw.Line(this, "tag1", true, 10, Low[0], 0, Brushes.Red); |     Casting a member of the **DrawObjects[]**collection must be done safely using the "as" keyword, otherwise you may receive exceptions at run time should another instance of the object (e.g., matching tag) exist from another owner:     | ns | | --- | | NinjaScript.DrawingTools.Line myLine = DrawObjects["tag1"] as DrawingTools.Line; |     **DrawingTools** anchor fields such as "Time" or "Price", etc have been moved to a **ChartAnchor** object owned by the drawing tool, rather than a direct field on the drawing object interface.  Please refer to the NinjaTrader 8 documentation for specific changes for each drawing tool:     | ns | | --- | | double linePrice = myLine.StartAnchor.Price; |     Objects which previously used**System.Drawing.Font** now uses new **NinjaTrader.Gui.Tools.SimpleFont** class:     | ns | | --- | | Gui.Tools.SimpleFont myFont = new Gui.Tools.SimpleFont("Arial", 12); |     Properties and other methods/objects which previously [System.Drawing.Color](https://msdn.microsoft.com/en-us/library/system.drawing.color(v=vs.110).aspx) structure now use the [System.Windows.Media.Brushes](https://msdn.microsoft.com/en-us/library/system.windows.media.brushes%28v=vs.110%29.aspx) class:     | ns | | --- | | BackBrush = Brushes.Blue; |      |  | | --- | | **Note**:  For custom **Brush** objects, it is important to .**Freeze()** the **Brush** due to the multi-threaded architecture of NinjaTrader 8.  Please be sure to review the new information on using [Brushes](https://ninjatrader.com/es/support/helpGuides/nt8/brushes.htm) |     **Namespaces**  The NinjaTrader 7 namespaces**NinjaTrader.Indicator** and **NinjaTrader.Strategy**have been renamed and moved to single **NinjaTrader.NinjaScript** namespace     | ns | | --- | | //This namespace holds indicators in this folder and is required. Do not change it. namespace NinjaTrader.NinjaScript.Indicators {   public class MyCustomIndicator : Indicator   {   } }   //This namespace holds Strategies in this folder and is required. Do not change it. namespace NinjaTrader.NinjaScript.Strategies {   public class MyCustomStrategy : Strategy   {   } } |     **Partial Classes (Porting methods and properties from UserDefinedMethods.cs)**  NinjaTrader 7 used a "UserDefinedMethods" class to define methods to be used across multiple NinjaScript indicators or strategies. In NinjaTrader 8, these pre-built partial classes have been removed to reduce a number of issues which could result from users sharing their UserDefinedMethods.cs files, or overwriting their existing files with copies from a new vendor. Partial classes are now best built manually and saved in the C:\Users\<user>\Documents\NinjaTrader 8\bin\Custom\AddOns folder.     |  | | --- | | **Warning**: If a partial class is saved in one of the folders used for specific NinjaScript objects other than AddOns (e.g., Indicators folder), auto-generated NinjaScript code may be appended to the end of the class by the NinjaScript Editor when compiled, which will cause a compilation error.  Saving these files in the AddOns folder will ensure they are still accessible and will not generate code which may be cause conflicts. |     You can use the template below as a starting point to create your partial class. If your partial class needs to inherit from a parent class, you can append the name of your desired parent class after the " : " to change the inheritance.     |  | | --- | | **Note**: Methods within your partial classes should be using the "public" modifier. |      | ns**Partial Class Example Template** | | --- | | namespace NinjaTrader.NinjaScript.Indicators {   public partial class MyMethods *// : parent class to inherit from*   {       //Sample method which calculates the delta of two prices       public double calculateDelta(double firstPrice, double secondPrice)       {           return Math.Abs(firstPrice - secondPrice);       }         //Sample method which prints Position information       public void printPositionInfo(Position position)       {           Print(String.Format("{0}: {1} {2} at {3}", position.Instrument, position.Quantity, position.MarketPosition, position.AveragePrice));       }           } } |     Below is an example of using one of the methods in this partial class from within an Indicator:     | ns**Partial Class Usage** | | --- | | protected override void OnBarUpdate() {   if (CurrentBar < 1) return;     // Use the static calculateDelta method to calculate the difference between the close of each bar   double delta = MyMethods.calculateDelta(Close[0], Close[1]);     Print(delta); } |      |  | | --- | | **Tip**:  At the time of the Beta implementation, the NinjaScript Editor does **NOT** include a partial class generator wizard, as it does for core NinjaScript Types such as Drawing Tools, Market Analyzer Columns, or Strategies. However, we are currently tracking a suggestion to implement a wizard for partial classes, under ID # **SFT-341**.   Please feel free to contact platformsupport@ninjatrader.com if you would like to add your vote for this enhancement. |     **Prevention of Redundant Data Loading**  In NinjaTrader 7, multiple Data Series could be added within a script, such as an indicator, and that script could then be hosted by another script, such as a strategy. While this is still possible in NinjaTrader 8, there is a new safeguard in place to prevent redundant data loading in both the hosting script and the hosted indicator.    When hosting an indicator which adds Data Series programmatically, the hosting script must include the same calls to the AddDataSeries() method as the hosted script. Without this, an error will result, which reads *"A hosted indicator tried to load additional data. All data must first be loaded by the hosting NinjaScript in its Configure state."* Without this safegaurd in place, it would be possible for unnecessarily large amounts of data to be loaded concurrently, as would be the case in a direct call to an indicator method on each OnBarUpdate(). By adding the calls to AddDataSeries() to the hosting script, you can ensure that the data is loaded when needed. Also, when this is done in the hosting script, all identical calls to AddDataSeries() in the hosted script will be ignored, as the data is already available.    The examples below show this in action:     | ns**Hosted Indicator Loads Additional Data** | | --- | | public class MyCustomIndicator : Indicator {   protected override void OnStateChange()   {     if (State == State.Configure)     {           AddDataSeries("AAPL", BarsPeriodType.Day, 1);           AddDataSeries("EURUSD", BarsPeriodType.Minute, 15);       }   } } |      | ns**Hosting Strategy Mirrors AddDataSeries() calls** | | --- | | public class MyCustomStrategy : Strategy {   // Define a MyCustomIndicator   MyCustomIndicator myIndicator;     protected override void OnStateChange()   {     if (State == State.Configure)     {         // Instantiate the MyCustomIndicator and add it to the chart         myIndicator = MyCustomIndicator();         AddChartIndicator(myIndicator);           // These calls to AddDataSeries() mirror the calls in the hosted indicator         AddDataSeries("AAPL", BarsPeriodType.Day, 1);         AddDataSeries("EURUSD", BarsPeriodType.Minute, 15);     }   } } |     **Bars with 0 Volume**  In previous versions, the NinjaTrader core was designed to replace a tick with a volume of 0 with a volume of 1.  This resulted in all ticks having a volume value of at least 1.  NinjaTrader 8 has removed that design policy and will now allow ticks with a volume of 0 to be processed.  This policy change may require logic changes to any custom bar types, indicators, or strategies which may have previously assumed volume would always be greater than 0.    **Multi-Series default "Trading Hours" templates**  The default behavior in NinjaTrader 8 will ensure that a bars series added to a script using [AddDataSeries()](https://ninjatrader.com/es/support/helpGuides/nt8/adddataseries.htm) will use the same "[TradingHours](https://ninjatrader.com/es/support/helpGuides/nt8/tradinghours.htm)" template as the primary series configured by the user. In contrast, the NinjaTrader 7 behavior was highly dependent on a number of variables.  We have updated this behavior to help with consistences and synchronization issues between multiple series; however if you your script relies on two times frames using different trading hours templates, you may consider using one of the new **tradingHours**string overloaded used in [AddDataSeries()](https://ninjatrader.com/es/support/helpGuides/nt8/adddataseries.htm):     | ns | | --- | | protected override void OnStateChange() {   if (State == State.Configure)   {     // adds a 1 minute AAPL bars with a default 24/7 session tempalte.     AddDataSeries("AAPL", new BarsPeriod { BarsPeriodType = BarsPeriodType.Minute, Value = 1 }, "Default 24 x 7");   } } |     **Miscellaneous**  All of the NinjaTrader 7 reference samples posted in our support forum have been updated to demonstrate NinjaTrader 8 functionality.  Please be sure to check the reference sample section to see other undocumented features and concepts which may not have been covered in the help guide:    [Official NinjaScript reference code samples](http://www.ninjatrader.com/support/forum/forumdisplay.php?f=30)    There are several other changes to implementation which are not covered in detail on this overview, please see the code breaking changes table at the bottom of this page which will compare the implementation changes between both versions. |

[permalink](https://ninjatrader.com/es/support/helpGuides/nt8/index.html?code_breaking_changes.htm#ImplementationChangesOverview)

tog_minus        [Signature Changes Overview](javascript:HMToggle('toggle','SignatureChangesOverview','SignatureChangesOverview_ICON'))

|  |  |
| --- | --- |
| **Signature**  A large number of the NinjaTrader methods which were available in NinjaTrader 7 have remained largely the same and should not generate any errors on compilation.  However there are a handful of existing methods signatures which have been updated in NinjaTrader 8 in order to fit within new framework which you would need to be aware of in order to transfer these functions from NinjaTrader 7 to NinjaTrader 8.  In most cases, the fundamental argument type has been restructured, which may result in compile errors depending on the type of object that is being used within the methods signature.     |  | | --- | | **Tip**:  Methods may now have additional signatures which add functionality which was not previously available.  Be sure to check the NinjaTrader 8 documentation which will cover all the available signatures available. | |

[permalink](https://ninjatrader.com/es/support/helpGuides/nt8/index.html?code_breaking_changes.htm#SignatureChangesOverview)

tog_minus        [Name Changes Overview](javascript:HMToggle('toggle','NameChangesOverview','NameChangesOverview_ICON'))

|  |
| --- |
| **Renamed**  During the NinjaTrader 8 development process, one of our goals to make sure that our core framework matched various coding standards which have been set out in the industry.  As a result of meeting these coding standards, many NinjaTrader methods and properties needed to been renamed.    While the functionality of these methods and properties remains the same, we chose to rename these variables to follow a semantically context specific naming convention which is generally agreed upon to favor readability.  We feel that the renaming of these properties and methods more explicitly describes the intended function to the developer who may be reviewing code.  The largest number of changes is in response to the name convention of bools, where they now follow a more strict verb-adjective or verb-noun structure.    For an example:    •The property **FirstTickOfBar** may have been hard to distinguish precisely what it represented without having to look up documentation.  In NinjaTrader 8, this property has been renamed to **IsFirstTickOfBar**, which now gives this property a more readable identifier name when you read this line of code as "*is the first tick of bar true?*"  •Another example is the case of **BarsSinceEntry()** which was renamed to**BarsSinceEntryExecution()**, which now specifies that this method is looking for an entry *execution*.  •NinjaTrader 7 sometimes had methods or properties which shared names, but references different data or actions.  For example **Add()** could have been used in reference to adding **DataSeries** to a script, adding a **Plot**, or adding a **Line**.  To be more specific, NinjaTrader 8 has renamed these to **AddDataSeries()**,**AddPlot()**, and **AddLine()** respectively.  •There may be cases where the property or method name has changed simply because the type of data it interacted with has changed.  (e.g., **BarColor** vs. **BarBrush**)  •There are other cases where properties may have used unnecessary brevity and was renamed to favor readability (e.g., **AvgPrice** vs **AveragePrice**)    These are just a few examples of the many name changes found in NinjaTrader 8 and some of the rational behind the number of these changes.  For simplicity, you will find a list of all the renamed properties in the table at the bottom of this document by filtering by the "Renamed" keyword. |

[permalink](https://ninjatrader.com/es/support/helpGuides/nt8/index.html?code_breaking_changes.htm#NameChangesOverview)

**Code Breaking Table**

Below you will find a reference table which lists all of the supported NinjaScript changes between NinjaTrader 7 and NinjaTrader 8.

Show 255075100All entries

Filter results:

| **Category** | **Base** | **NT7 Method/Property** | **NT8 Method/Property** | **Summary** |  |
| --- | --- | --- | --- | --- | --- |
| Implementation | Strategy | [GetAccountValue()](http://www.ninjatrader.com/support/helpGuides/nt7/index.html?getaccountvalue.htm) | [Account.Get()](http://www.ninjatrader.com/support/helpGuides/nt8/en-us/index.html?get.htm) | Access to Account values have been directly exposed | Details |
| Renamed | Strategy | [Add() - Strategy](http://www.ninjatrader.com/support/helpGuides/nt7/index.html?add2.htm) | [AddChartIndicator()](http://www.ninjatrader.com/support/helpGuides/nt8/en-us/index.html?addchartindicator.htm) | Method renamed to be more specific |  |
| Implementation | General | [Add() - Data](http://www.ninjatrader.com/support/helpGuides/nt7/index.html?add3.htm) | [AddDataSeries()](http://www.ninjatrader.com/support/helpGuides/nt8/en-us/index.html?adddataseries.htm) | Method was renamed to be more specific, received a number of enhancements. | Details |
| Signature | General | [AddKagi()](http://www.ninjatrader.com/support/helpGuides/nt7/index.html?addkagi.htm) | [AddKagi()](http://www.ninjatrader.com/support/helpGuides/nt8/en-us/index.html?addkagi.htm) | Received a number of signature changes | Details |
| Renamed | Indicator | [Add() - Line](http://www.ninjatrader.com/support/helpGuides/nt7/index.html?add.htm) | [AddLine()](http://www.ninjatrader.com/support/helpGuides/nt8/en-us/index.html?addline.htm) | Method renamed to be more specific |  |
| Signature | General | [AddLineBreak()](http://www.ninjatrader.com/support/helpGuides/nt7/index.html?addlinebreak.htm) | [AddLineBreak()](http://www.ninjatrader.com/support/helpGuides/nt8/en-us/index.html?addlinebreak.htm) | Received a number of signature changes | Details |
| Renamed | Indicator | [Add() - Plot](http://www.ninjatrader.com/support/helpGuides/nt7/index.html?add.htm) | [AddPlot()](http://www.ninjatrader.com/support/helpGuides/nt8/en-us/index.html?addplot.htm) | Method renamed to be more specific |  |
| Signature | General | [AddPointAndFigure()](http://www.ninjatrader.com/support/helpGuides/nt7/index.html?addpointandfigure.htm) | [AddPointAndFigure()](http://www.ninjatrader.com/support/helpGuides/nt8/en-us/index.html?addpointandfigure.htm) | Received a number of signature changes | Details |
| Signature | General | [AddRenko()](http://www.ninjatrader.com/support/helpGuides/nt7/index.html?addrenko.htm) | [AddRenko()](http://www.ninjatrader.com/support/helpGuides/nt8/en-us/index.html?addrenko.htm) | Received a number of signature changes | Details |
| Signature | General | [Alert()](http://www.ninjatrader.com/support/helpGuides/nt7/index.html?alert.htm) | [Alert()](http://www.ninjatrader.com/support/helpGuides/nt8/en-us/index.html?alert.htm) | Color no longer used, use Brushes instead; soundLocation now requires absolute file path | Details |
| Implementation | Drawing | [IAndrewsPitchfork](http://www.ninjatrader.com/support/helpGuides/nt7/index.html?iandrewspitchfork.htm) | [AndrewsPitchfork](http://www.ninjatrader.com/support/helpGuides/nt8/en-us/index.html?andrewspitchfork.htm) | IDrawingObjects have been replaced | Details |
| Implementation | Drawing | [IArc](http://www.ninjatrader.com/support/helpGuides/nt7/index.html?iarc.htm) | [Arc](http://www.ninjatrader.com/support/helpGuides/nt8/en-us/index.html?arc.htm) | IDrawingObjects have been replaced | Details |
| Renamed | Indicator | [LinesConfigurable](http://www.ninjatrader.com/support/helpGuides/nt7/index.html?linesconfigurable.htm) | [AreLinesConfigurable](http://www.ninjatrader.com/support/helpGuides/nt8/en-us/index.html?arelinesconfigurable.htm) | Property renamed to meet naming conventions |  |
| Renamed | Indicator | [PlotsConfigurable](http://www.ninjatrader.com/support/helpGuides/nt7/index.html?plotsconfigurable.htm) | [ArePlotsConfigurable](http://www.ninjatrader.com/support/helpGuides/nt8/en-us/index.html?areplotsconfigurable.htm) | Property renamed to meet naming conventions |  |
| Implementation | Drawing | [IArrowDown](http://www.ninjatrader.com/support/helpGuides/nt7/index.html?iarrowdown.htm) | [ArrowDown](http://www.ninjatrader.com/support/helpGuides/nt8/en-us/index.html?arrowdown.htm) | IDrawingObjects have been replaced | Details |
| Implementation | Drawing | [IArrowLine](http://www.ninjatrader.com/support/helpGuides/nt7/index.html?iarrowline.htm) | [ArrowLine](http://www.ninjatrader.com/support/helpGuides/nt8/en-us/index.html?arrowline.htm) | IDrawingObjects have been replaced | Details |
| Implementation | Drawing | [IArrowUp](http://www.ninjatrader.com/support/helpGuides/nt7/index.html?iarrowup.htm) | [ArrowUp](http://www.ninjatrader.com/support/helpGuides/nt8/en-us/index.html?arrowup.htm) | IDrawingObjects have been replaced | Details |
| Implementation | General | [DataSeries.Set()](http://www.ninjatrader.com/support/helpGuides/nt7/index.html?dataseries_class.htm) | [Assignment Operator (=)](http://www.ninjatrader.com/support/helpGuides/nt8/en-us/index.html?seriest.htm) | The .Set() method has been replaced | Details |
| Implementation | Strategy | [AtmStrategyCreate()](http://www.ninjatrader.com/support/helpGuides/nt7/index.html?atmstrategycreate.htm) | [AtmStrategyCreate()](http://www.ninjatrader.com/support/helpGuides/nt8/en-us/index.html?atmstrategycreate.htm) | Added a callback signature parameter | Details |
| Renamed | Strategy | [AvgBarsInTrade](http://www.ninjatrader.com/support/helpGuides/nt7/index.html?avgbarsintrade.htm) | [AverageBarsInTrade](http://www.ninjatrader.com/support/helpGuides/nt8/en-us/index.html?averagebarsintrade.htm) | Property renamed to favor readability |  |
| Renamed | Strategy | [AvgEtd](http://www.ninjatrader.com/support/helpGuides/nt7/index.html?avgetd.htm) | [AverageEtd](http://www.ninjatrader.com/support/helpGuides/nt8/en-us/index.html?averageetd.htm) | Property renamed to favor readability |  |
| Renamed | Strategy | [AvgMae](http://www.ninjatrader.com/support/helpGuides/nt7/index.html?avgmae.htm) | [AverageMae](http://www.ninjatrader.com/support/helpGuides/nt8/en-us/index.html?averagemae.htm) | Property renamed to favor readability |  |
| Renamed | Strategy | [AvgMfe](http://www.ninjatrader.com/support/helpGuides/nt7/index.html?avgmfe.htm) | [AverageMfe](http://www.ninjatrader.com/support/helpGuides/nt8/en-us/index.html?averagemfe.htm) | Property renamed to favor readability |  |
| Renamed | Strategy | [AvgPrice](http://www.ninjatrader.com/support/helpGuides/nt7/index.html?avgprice.htm) | [AveragePrice](http://www.ninjatrader.com/support/helpGuides/nt8/en-us/index.html?position_averageprice.htm) | Property renamed to favor readability |  |
| Renamed | Strategy | [AvgProfit](http://www.ninjatrader.com/support/helpGuides/nt7/index.html?avgprofit.htm) | [AverageProfit](http://www.ninjatrader.com/support/helpGuides/nt8/en-us/index.html?averageprofit.htm) | Property renamed to favor readability |  |

Showing 1 to 25 of 197 entries

Previous12345…8Next

[permalink](https://ninjatrader.com/es/support/helpGuides/nt8/index.html?code_breaking_changes.htm#codebreakingtable)

|  |  |
| --- | --- |
| **Navigation:**  [NinjaScript](https://ninjatrader.com/es/support/helpGuides/nt8/ninjascript.htm) >  **NinjaScript Best Practices** | [Previous page](https://ninjatrader.com/es/support/helpGuides/nt8/code_breaking_changes.htm) [Return to chapter overview](https://ninjatrader.com/es/support/helpGuides/nt8/ninjascript.htm) [Next page](https://ninjatrader.com/es/support/helpGuides/nt8/distribution.htm) |

There are some best practices to be aware of when developing NinjaScript classes. The following tables present a non-exhaustive list of considerations to keep in mind when designing and implementing your code.

|  |
| --- |
| **Note**:   NinjaTrader is multi-threaded and event driven. Always assume that any of the methods you implement in NinjaScript could be called from another thread. |

tog_minus        [State management practices](javascript:HMToggle('toggle','StateResourceManagement','StateResourceManagement_ICON'))

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Managing Resources**  The [OnStateChange(](https://ninjatrader.com/es/support/helpGuides/nt8/onstatechange.htm)) method is called anytime there has been a change of [State](https://ninjatrader.com/es/support/helpGuides/nt8/state.htm) and can be used to help you setup, manage, and destroy several types of resources.  Where these values are setup is highly dependent on the kind of resource you are using.  The section below will cover how to manage various resources throughout different states.    **Setting Default UI Property Grid values**  Reserve **State.SetDefaults** for defaulting any public properties you wish to have exposed on the UI property grid.   You should also use this State for setting default desired NinjaScript property behavior which can be overridden from the property grid (e.g. [Calculate](https://ninjatrader.com/es/support/helpGuides/nt8/calculate.htm), [IsOverlay](https://ninjatrader.com/es/support/helpGuides/nt8/isoverlay.htm), etc.).  For Plots and Lines you wish to configure, [AddPlot()](https://ninjatrader.com/es/support/helpGuides/nt8/addplot.htm), [AddLine()](https://ninjatrader.com/es/support/helpGuides/nt8/addline.htm) should also have their default values set during this State     |  | | --- | | **Why:**Public values of the NinjaScript object in **SetDefaults** are pushed to the UI property grid for an opportunity to change settings of your object. |      | ns **Best practice** | | --- | | protected override void OnStateChange() {   // these are the values that show up as default on the UI   if (State == State.SetDefaults)   {     Calculate = Calculate.OnPriceChange;     IsOverlay = false;       Period = 50;       AddPlot(Brushes.Blue, "Plot Value");     AddLine(Brushes.Gray, 100, "Threshold");   } } |     For public properties you do **NOT** wish exposed to the UI property grid, set the [Browsable](https://ninjatrader.com/es/support/helpGuides/nt8/browsableattribute.htm) attribute to false:     | ns **Best practice** | | --- | | [Browsable(false)] // prevents from showing up on the UI property grid public int Communicator { get; set; } |     On indicators, properties you wish to set from other objects, set the [NinjaScriptPropertyAttribute](https://ninjatrader.com/es/support/helpGuides/nt8/ninjascriptpropertyattribute.htm):     | ns **Best practice** | | --- | | [NinjaScriptProperty] // can now call MyIndicator(20) from another object public int Period { get; set; } |     The default behavior is to serialize any public properties to a Workspace or Template file when saving.  However, not all objects can be serialized - or you may wish to exclude a property from being saved and restored.  For these scenarios, set the [XMLIgnore](https://ninjatrader.com/es/support/helpGuides/nt8/xmlignoreattribute.htm) attribute to the property:     | ns **Best practice** | | --- | | [XmlIgnore] // removes from serialization     public Brush DownBrush { get; set; } |      |  | | --- | | **Tip:**See the [Working with Brushes](https://ninjatrader.com/es/support/helpGuides/nt8/working_with_brushes.htm) section of the Help Guide for information on properly serializing brushes |     **Calculating run-time object values** Do not attempt to do advanced calculations or try to access object references in **State.SetDefaults**.  This State should be kept as lean as possible, and any calculation logic should be delayed until at least **State.Configure**     |  | | --- | | **Why:**Your object will be called in situations you may not be expecting. You can read more about this subject on [Understanding the life cycle of your NinjaScript objects](https://ninjatrader.com/es/support/helpGuides/nt8/understanding_the_lifecycle_of.htm) |      | ns **Practice to avoid** | | --- | | protected override void OnStateChange() {   if (State == State.SetDefaults)   {       // logic could take longer than desired as the list of indicator names is populated     for (int i = 0; i <= array.length; i ++)         DoWork(i);      // possible null reference exception since TickSize is not set yet     Period = 5 \* TickSize;   } } |      | ns **Best practice** | | --- | | protected override void OnStateChange() {   // Complex operations should be delayed to >= State.Configure   if (State == State.Configure)   {     for (int i = 0; i < = array.length; i ++)         DoWork(i);   }     // information related to market data is not available until at least State.DataLoaded   else if (State == State.DataLoaded)   {     Period = 5 \* TickSize;   } } |     **Setting class level variables**  Do not set variables at the class level unless they are constant.  You should delay setting or resetting variables until the **State** has reached **State.Configure**.  You can use const keyword to differentiate values which do not change from variables which do change.     |  | | --- | | **Why**:  Waiting to set up and define resources until the object has been configured ensures that values not set up and declared prematurely. |      | ns **Best practice** | | --- | | // value is always 5, it can be made constant and declared at the class level private const int multiplier = 5;   // these values can change, may be better to delay setting until State.Configure private int counter; private List<int> myList;  protected override void OnStateChange() {   if (State == State.Configure)   {     counter = 0;     myList = new List<int>();   }   } |     **Resetting class level variables for Strategy Analyzer Optimization**    To take advantage of performance optimizations, developers may need to reset class level variables in the strategy otherwise unexpected results can occur.     |  | | --- | | **Why**:  When optimizing a strategy, instances may or may not be recycled depending on the strategy [IsInstantiatedOnEachOptimizationIteration](https://ninjatrader.com/es/support/helpGuides/nt8/isinstantiatedoneachoptimizationiteration.htm) setting. |      | ns **Best practice** | | --- | | // examples of fields which need to be reset private double myDouble; private bool myBool; private DateTime myDateTime; private Order myOrderObject; private Brush myBrushObject; private Array myIntArray; private List<object> myList; private SMA mySMAIndicator; private Series<double> mySeries;   protected override void OnStateChange() {   if (State == State.SetDefaults)   {     // disabled to take advantage of performance gains     // However any strategy state that would be mutable after State.SetDefaults needed to be reset for the next run.     IsInstantiatedOnEachOptimizationIteration = false;   }   else if (State == State.Configure)   {     // Since these values are not dependent on bars, they can be reset as early as State.Configure     myDouble = double.MinValue;     myBool = false;     myDateTime = DateTime.MinValue;     myOrderObject = null;     myBrushObject = null;       if (myIntArray != null)         Array.Clear(myIntArray, 0, myIntArray.Length);     else         myIntArray = new int[20];       if (myList != null)         myList.Clear();     else         myList = new List<object>();   }     else if (State == State.DataLoaded)   {     // Since these values do are dependent on bars, they should only reset during State.DataLoaded     mySMAIndicator = SMA(14);     mySeries = new Series<double>(this);   } } |     **Accessing properties related to market data**  Do not attempt to access objects related to instrument market data until the **State** has reached **State.DataLoaded**     |  | | --- | | **Why**: Waiting to access objects that depend on market data until **DataLoaded** prevents access errors in all scenarios |      | ns **Best practice** | | --- | | protected override void OnStateChange() {   if (State == State.DataLoaded)   {     // these objects and their related members are not available until State.DataLoaded     Print(Bars.Count);     Print(Instrument.FullName);     Print(BarsPeriod.BarsPeriodType);     Print(TradingHours.TimeZone);     Print(Input);   } } |      |  | | --- | | **Note**: All additional data series must be added in **State.Configure**(this includes series that any hosted script potentially needs as well - [more info](http://ninjatrader.com/support/helpGuides/nt8/en-us/adddataseries.htm)). Since objects such as [Instrument](https://ninjatrader.com/es/support/helpGuides/nt8/instrument.htm), [BarsPeriod](https://ninjatrader.com/es/support/helpGuides/nt8/barsperiod.htm), [TradingHours](https://ninjatrader.com/es/support/helpGuides/nt8/tradinghours.htm), etc. are **NOT** guaranteed to be available until **State.DataLoaded**, you cannot reliably use the primary instrument properties as arguments in [AddDataSeries()](https://ninjatrader.com/es/support/helpGuides/nt8/adddataseries.htm).  Attempting to add a data series dynamically is **NOT** guaranteed and therefore should be avoided.  In some cases, you may be able to use a [BarsRequest()](https://ninjatrader.com/es/support/helpGuides/nt8/barsrequest.htm) to obtain market data for other instruments and intervals. |     **Setting up resources that rely on market data**  For objects which depend on market data, delay their construction until the **State** has reached **State.DataLoaded**     |  | | --- | | **Why**: Waiting to construct objects that depend on market data until **DataLoaded** ensures that their underlying input contains significant values in all scenarios. |      | ns **Best practice** | | --- | | // these resources depend on bars, wait until State.DataLoaded to instantiated private EMA myEMA; private Series<double> mySeries; private SessionIterator mySessionIterator;   protected override void OnStateChange() {     if (State == State.DataLoaded)   {     myEMA = EMA(20);     mySeries = new Series<double>(this);     mySessionIterator = new SessionIterator(Bars);   } } |     **Accessing element on the UI**  For objects which exist on the UI (e.g., [ChartControl](https://ninjatrader.com/es/support/helpGuides/nt8/chartcontrol.htm), [ChartPanel](https://ninjatrader.com/es/support/helpGuides/nt8/chartpanel.htm), [ChartBars](https://ninjatrader.com/es/support/helpGuides/nt8/chartbars.htm), [NTWindow](https://ninjatrader.com/es/support/helpGuides/nt8/ntwindow.htm), etc.) wait until the State has reached State.Historical.  This practice is correct for both reading properties or should you wish to add custom elements to the existing UI.     |  | | --- | | **Why**:  NinjaTrader UI related objects are not guaranteed to be available until historical data processing has started. |      | ns **Best practice** | | --- | | protected override void OnStateChange() {   // wait until at least State.Historical   if (State == State.Historical)   {     // and double check UI object is not null before accessing     if (ChartControl != null)     {         Print(ChartControl.Properties.ChartBackground);     }   } } |     **Transitioning order references from historical to real-time**  When dealing with strategy based orders which have transitioned from historical to real-time, you will need to ensure that locally stored order references are also updated.     |  | | --- | | **Why**: As the core order object updates, NinjaTrader has no specific way to update your locally stored order references.  You can read more about this subject on the Advanced Order Handling topic: [Transitioning order references from historical to live](https://ninjatrader.com/es/support/helpGuides/nt8/advanced_order_handling.htm) |      | ns **Best practice** | | --- | | protected override void OnStateChange() {   // one time only, as we transition from historical to real-time   if (State == State.Realtime)   {     // convert any old historical order object references     // to the new live order submitted to the real-time account     if (myOrder != null)         myOrder = GetRealtimeOrder(myOrder);   } } |     **Terminating custom resources**  Use a flag to track when resources have been set up properly before attempting to destroy them.     |  | | --- | | **Why**:  Checking that an object has been configured ensures that values not destroyed prematurely. You can read more about this subject on [Understanding the life cycle of your NinjaScript objects](https://ninjatrader.com/es/support/helpGuides/nt8/understanding_the_lifecycle_of.htm) |      | ns **Best practice** | | --- | | protected override void OnStateChange() {   if (State == State.Configure)   {     myObject = new object();     // set a flag to indicator object has been configured     configured = true;   }     else if (State == State.Terminated)   {     // only dispose of object if it has been configured     if (configured)     {         myObject.Dispose();     }   } } | |

[permalink](https://ninjatrader.com/es/support/helpGuides/nt8/index.html?ninjascript_best_practices.htm#StateResourceManagement)

tog_minus        [Error handling practices](javascript:HMToggle('toggle','Errorhandling','Errorhandling_ICON'))

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Safely accessing reference objects**  Although there are documented **States** where objects are available, the implementation could change.  If you are accessing a reference object, please do so by first checking that the object is not null.     | ns **Best practice** | | --- | | // checking to ensure chart control is available in all situations // will help to ensure this logic below does not generate errors at a later time if(ChartControl != null) {   myBackgroundBrush = ChartControl.Properties.ChartBackground; } |     **Accessing objects which terminate**  To protect against race conditions and access errors, you should temporarily check for reference errors any time you attempt to do something with an object.     |  | | --- | | **Why**: **OnStateChange()** runs asynchronous to other NinjaScript events.  You can run into scenarios where you **State.Terminated** logic is called in the middle of OnBarUpdate(), OnRender() etc. |      | ns **Best practice** | | --- | | protected override void OnStateChange() {   // this logic runs asynchronously to other events   if (State == State.Terminated)   {     myObject = null;   } } protected override void OnRender(ChartControl chartControl, ChartScale chartScale) {   if (myObject == null)     return;     // for safety, always check for null references before attempting to access an object   // even if you have once checked for null references earlier run-time   if (myObject != null)     myObject.DoSomething(); } |     **Proving instructions for non-ninjascript properties**  Do not attempt to modify existing UI "Properties" to meet your specific needs.  These features are exposed to allow you to read the environment state and make decisions to alter how your code executes, but should not be relied on to modify settings on behalf of the user.  While these objects from these classes have setters for technical reasons, you should not attempt to amend the values through code.  Instead, you should issue warnings or log errors instructing users to modify settings when required:     |  | | --- | | **Why**:  NinjaTrader makes no guarantee that the requested changes will take effect, and user settings always take precedences.  This includes the user defined [ChartControl.Properties](https://ninjatrader.com/es/support/helpGuides/nt8/chartcontrol_properties.htm), [ChartBars.Properties](https://ninjatrader.com/es/support/helpGuides/nt8/chartbars_properties.htm), and [ChartPanel.Properties](https://ninjatrader.com/es/support/helpGuides/nt8/chartscale_properties.htm). Furthermore, two different user scripts could be installed which also attempt to modify properties you are relying which could introduce conflicts. |      | ns **Best practice** | | --- | | if (State == State.Historical) {   if (ChartControl.Properties.EquidistantBarSpacing == true)   {     Draw.TextFixed(this, "error", "This indicator works best with Equidistant BarSpacing set to false.", TextPosition.BottomRight);   } } |     **Modifying UI elements and multi-threading**  When interacting with UI objects, such as obtaining UI information, or modifying the existing layout, always use the NinjaScript's Dispatcher asynchronously     |  | | --- | | **Critical**:  Improper thread handling from a NinjaScript object is a common cause of application deadlocks.  Please be sure to read more information on [Multi-Threading Consideration for NinjaScript](https://ninjatrader.com/es/support/helpGuides/nt8/multi-threading.htm) |      | ns **Best practice** | | --- | | // using a Dispatcher will ensure that the corresponding action executes on the associated thread this.Dispatcher.InvokeAsync(() => {   UserControlCollection.Add(new System.Windows.Controls.TextBlock   {     Text = "\nAdded by the ChartControl Dispatcher."   }); }); |       **Properly implementing try/catch blocks**  Unless you are specifically debugging a method, the use of a try-catch block should be scoped to a particular area of logic.  Do **NOT** try to handle all of your execution logic under one giant try-catch block.     |  | | --- | | **Why**:  Larger try-catch blocks can not only be harder to debug, but can introduce performance issues at run-time |      | ns **Practice to avoid** | | --- | | protected override void OnBarUpdate() {   try   {     // encapsulates entire OnBarUpdate logic   }   catch (Exception ex)   {     // attempt to handle all errors in one catch   } } |     **Using WPF brushes**  Try to use a static predefined Brush if possible.  If you need to customize a new brush object, make sure to .Freeze() the brush before using it.     |  | | --- | | **Why**:  The pre-defined brushes are thread safe and do not require any special handling.  Custom defined brushes, on the other hand, are **NOT** thread-safe and must be frozen otherwise cross-thread exceptions can occur. |      | ns **Best practice** | | --- | | // predefined brush BackBrush = Brushes.Blue;   // if you are using a custom brush to e.g., modify the opacity SolidColorBrush opaqueBlue = new SolidColorBrush(Colors.Blue) {Opacity = .25f};   // or just using at custom color not available in pre-defined brushes class SolidColorBrush coolGreen = new SolidColorBrush(Color.FromRgb(30, 255, 128));   // you must freeze these brushes after they are constructed! opaqueBlue.Freeze(); coolGreen.Freeze(); |     **barsAgo indexer vs. absolute bar Index**  As you probably know, you can quickly look up the bar value on the chart by calling a [PriceSeries<T>](https://ninjatrader.com/es/support/helpGuides/nt8/priceseries.htm) barsAgo indexer, e.g., Close[0].  However, the internal indexer and pointers about the barsAgo value are only guaranteed to be correctly synced and updated during a market data event.  As a result, you should favor using the absolute [GetValueAt()](https://ninjatrader.com/es/support/helpGuides/nt8/getvalueat.htm) methods during events which are not driven by price     |  | | --- | | **Why**:  Attempting to call the barsAgo indexer in an event method that is not driven by market data can yield unexpected results. |      | ns **Best practice** | | --- | | // OnRender is not a market data event; barsAgo pointers are not guaranteed to be in sync protected override void OnRender(ChartControl chartControl, ChartScale chartScale) {   Print(mySMA.GetValueAt(CurrentBar)); }   // same is true for you custom events private void myCustomClickHandler(object sender, MouseButtonEventArgs e) {   Print(Close.GetValueAt(CurrentBar)); } |      |  | | --- | | **Tip**:  If you have programming requirements which rely on a PriceSeries indexer, you can use the [TriggerCustomEvent()](https://ninjatrader.com/es/support/helpGuides/nt8/triggercustomevent.htm) delegate which will update the internal pointers and indexes before executing the logic you specify. |     **Casting safely**  Avoid type casting and type conversion as much as possible.  Casting from a mixed collection of types is also prone to exceptions especially in situations that may not occur when you originally test your code.     |  | | --- | | **Why**:   The practice to avoid code below could work in some scenarios but would generate errors if other types were added to that collection that you were not anticipating. |      | ns **Practice to avoid** | | --- | | // This would run without errors if there were \_ONLY\_ type HoriztonalLine on the chart // But you risk a likely 'System.InvalidCastException' when other draw types are in that collection foreach (HorizontalLine hLine in DrawObjects) {   } |     If you must cast, do so safely and avoid implicit casts to types which may not be guaranteed to succeeded     | ns **Best practice** | | --- | | // Use the base IDrawingTool type and then cast to the desired type within the for loop foreach (IDrawingTool hLine in DrawObjects) {   // Note:  to prevent further errors, your type casting should be done using the "as" keyword   // Opposed to a direct cast:   // HorizontalLine myLine = (HorizontalLine) hLine;    HorizontalLine myLine = hLine as HorizontalLine;     // This will allow you to ensure the cast actually occurred   if (myLine != null)   {     Print(myLine.StartAnchor.Price);   } } | |

[permalink](https://ninjatrader.com/es/support/helpGuides/nt8/index.html?ninjascript_best_practices.htm#Errorhandling)

tog_minus        [Performance practices](javascript:HMToggle('toggle','Performance','Performance_ICON'))

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Referencing indicator methods**  In general, when calling an Indicator return method, there is some internal caching which occurs by design to help reduce memory conception.     |  | | --- | | **Why**:  While the designed indicator caching improves general memory performance, there is an implied cost of actually looking up the cached indicator |      | ns **Practice to avoid** | | --- | | // each time you call the SMA() return method there is a small performance cost // implied from the time it takes to look up the cached instance if (Close[0] > SMA(20)[0]) {   Print(SMA(20)[0]);   EnterLongLimit(SMA(20)[0]);   Draw.Dot(this, Time[0].ToString(), false, 0, SMA(20)[0], Brushes.DarkGreen); } |      |  | | --- | | **Note**:  Indicator caching **ONLY**occurs when an indicator is recalled with the same **EXACT** parameters and input. (i.e. when a previously called indicator is called a second time with new parameters, a second instance will be created / cached) |     If you are reusing an indicator several times through your code (especially indicators with many parameters), you can take further steps to refine performance by storing a reference to the indicator instance yourself (although it is by no means a requirement, and this suggestion does not need to be followed strictly)     | ns **Best practice** | | --- | | private SMA mySma;   protected override void OnStateChange() {   // when the indicator begins processing   // save an instance of the SMA indicator with the desired input     if (State == State.Historical)   {     mySma = SMA(20);   } }   protected override void OnBarUpdate() {   // use the referenced mySMA throughout the lifetime of the script   if (Close[0] > mySma[0])   {     Print(mySma[0]);     EnterLongLimit(mySma[0]);     Draw.Dot(this, Time[0].ToString(), false, 0, mySma[0], Brushes.DarkGreen);   } } |     **Marking object references for garbage collection**  While it is not always necessary to set objects to null, doing so will mark them for garbage collection sooner and help prevent unnecessary memory resources from being utilized.     |  | | --- | | **Why**:   In general you should be diligent to set stored memory objects to null when you are done using them, especially in situations where a NinjaScript object may be running for an extended period. |      | ns **Best practice** | | --- | | protected override void OnBarUpdate() {   // saving "myDot" creates an additional reference in memory   Dot myDot = Draw.Dot(this, "myDot" + CurrentBar, false, Time[0], Close[0], Brushes.Blue);     if (conditionToRemove)   {     // remove draw object will remove the object from the chart     RemoveDrawObject("myDot");       // but your local object "myDot" is still stored in memory.     // Explicitly setting to null will ensure object is marked for garbage collection     myDot = null;   } } |      |  | | --- | | **Note**:  The example above demonstrates using a draw object, but the practice can be extended to any object you store in memory (e.g., orders, brushes, custom objects, etc) |     **Disposing of custom resources**  Dispose of objects that inherit from IDisposable or put into a Using statement.     |  | | --- | | **Why**:  NinjaTrader is not guaranteed to dispose of objects for you.  To avoid unnecessary memory consumption, always manage your resources by creating a variable and dispose of the object. |      | ns **Best practice** | | --- | | // example of object instantiated which need to be disposed StreamWriter writer = new StreamWriter("some\_file.txt");   // use the object writer.WriteLine("Some text");   // implements IDisposbile, make sure to call .Dispose() when finished writer.Dispose();   // or put in "using" statement which implicitly calls .Dispose() when finished using (StreamWriter writer2 = new StreamWriter("some\_file.txt")) {   writer2.WriteLine("Some text"); } |      |  | | --- | | **Tip**:  This is most commonly applicable when using SharpDX resources for custom rendering.  Please be sure to review the information on [Best Practices for SharpDX Resources](https://ninjatrader.com/es/support/helpGuides/nt8/using_sharpdx_for_custom_chart_rendering.htm#bestpracticesforsharpdxresources) |     **Avoiding duplicate calculations**  Be mindful where and when your potentially complex calculations would be recalculated and thus run the risk of being calculated redundantly. For example, you may have logic which only needs to calculate, e.g., once per instance, once per session, once per bar, etc.     | ns **Best practice** | | --- | | // get GetPreviousTradingDayEnd() is expensive to look up // but value only needs to be looked up once a day -> only calcualte on first bar of session if (Bars.IsFirstBarOfSession) {   TradingHours.GetPreviousTradingDayEnd(Time[0]); } |     The same considerations would apply to variables or function calls that would not change their output value for the currently processed bar on [Calculate.OnEachTick](https://ninjatrader.com/es/support/helpGuides/nt8/calculate.htm) or [.OnPriceChange](https://ninjatrader.com/es/support/helpGuides/nt8/calculate.htm), thus there would be no need handling them outside of [IsFirstTickOfBar](https://ninjatrader.com/es/support/helpGuides/nt8/isfirsttickofbar.htm)     | ns **Best practice** | | --- | | // dedicated logic to cache the prior sum on each tick of bar // While it is a good practice, this can cause problems for bar types which may remove last bar (see below) if (IsFirstTickOfBar)   priorSum = sum;   sum = priorSum + Input[0] - (CurrentBar >= Period ? Input[Period] : 0); Value[0] = sum / (CurrentBar < Period ? CurrentBar + 1 : Period); |     **Caching values on bars which remove last bar**  Building on the previous example, be careful when caching values on the first tick of bar if using bars types which are [IsRemoveLastBarSupported](https://ninjatrader.com/es/support/helpGuides/nt8/isremovelastbarsupported.htm).  To see how to handle these situations best, take a look at the default SMA indicator which has an additional logic branch which disables caching on those bar types:     | ns **Best practice** | | --- | | // logic below disables first tick of bar caching only on bar types which remove last bar if (BarsArray[0].BarsType.IsRemoveLastBarSupported) {   if (CurrentBar == 0)     Value[0] = Input[0];   else   {     double last = Value[1] \* Math.Min(CurrentBar, Period);       if (CurrentBar >= Period)         Value[0] = (last + Input[0] - Input[Period]) / Math.Min(CurrentBar, Period);     else         Value[0] = ((last + Input[0]) / (Math.Min(CurrentBar, Period) + 1));   } } |     **Precomputing values instead of calculating in OnRender()**  To preserve good performance, always err on the side of caution if you are using OnRender for any calculation logic.     |  | | --- | | **Why**:   OnRender() is called frequently as you interact with the Chart, which can cause calculations to occur much more often than the related market data events and can cause unnecessary spikes in CPU consumption. |      | ns **Practice to avoid** | | --- | | protected override void OnRender(ChartControl chartControl, ChartScale chartScale) {  // continually recalling the same value methods is unnecessary in this situation       double myValue = Bars.GetClose(CurrentBar) + Bars.GetOpen(CurrentBar);     // render myValue } |      | ns **Best practice** | | --- | | private double myValue;  protected override void OnBarUpdate() {   // myValue only needs to update when OnBarUpdate() is called   // and then can be passed to OnRender() for chart rendering purposes   myValue = Close[0] + Open[0]; }   protected override void OnRender(ChartControl chartControl, ChartScale chartScale) {   // if needed, you can always check that myValue has actually been set   if (myValue > double.MinValue)   {     // render myValue   } } |     **Restricting OnRender() calculations to visible ChartBars**  Use the [ChartBars.FromIndex](https://ninjatrader.com/es/support/helpGuides/nt8/chartbars_fromindex.htm) and [ChartBars.ToIndex](https://ninjatrader.com/es/support/helpGuides/nt8/chartbars_toindex.htm) to limit calculations to only what is visible on the chart     |  | | --- | | **Why:**Rendering should be reserved for rendering on what is visible on the Chart.  Performing calculations on bar index which are not visible can cause random spikes in CPU consumption. |      | ns **Best practice** | | --- | | protected override void OnRender(ChartControl chartControl, ChartScale chartScale) {   // restricting this loop to only the ChartBars.From/ToIndex limits the loop to only what is visible on the chart   for (int barIndex = ChartBars.FromIndex; barIndex <= ChartBars.ToIndex; barIndex++)   {     Print(ChartControl.GetSlotIndexByX(barIndex));   } } |     **Using DrawObjects vs custom graphics in OnRender()**  When using [Draw methods](https://ninjatrader.com/es/support/helpGuides/nt8/drawing.htm), a new instance of the Draw object is created including its custom rendering and calculation logic.  These methods are convenient in many situations, but can quickly introduce performance issues if used too liberally.  In some situations, you may see better performance for rendering via [SharpDX](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx.htm) in [OnRender()](https://ninjatrader.com/es/support/helpGuides/nt8/onrender.htm).     |  | | --- | | **Why**: Each draw object instance will see its own OnRender() called to render values. If you instead implement custom rendering in the your object, you would only see a single OnRender() call for your custom created graphics. |      | ns **Practice to avoid** | | --- | | protected override void OnBarUpdate() {   // this would draw a dot on every bar on the chart   // each instance would need to call its own OnRender() method   // not a very efficient use a draw method   Draw.Dot(this, "everyDot" + CurrentBar, false, 0, Close[0], Brushes.Blue); } |     With just a little extra code (much less than what is in the Draw methods) custom SharpDX rendering greatly reduces CPU and Memory consumption     | ns **Best practice** | | --- | | protected override void OnRender(ChartControl chartControl, ChartScale chartScale) {   // achieves the same effect of drawing a dot on every bar   // but only needs to call your object's OnRender()   for (int index = ChartBars.FromIndex; index <= ChartBars.ToIndex; index++)   {     float price = chartScale.GetYByValue(Close.GetValueAt(index));     float bar = chartControl.GetXByBarIndex(ChartBars, index);     float radius = (float) chartControl.BarWidth;       SharpDX.Direct2D1.Ellipse dot = new SharpDX.Direct2D1.Ellipse(new SharpDX.Vector2(bar, price), radius, radius);       using (SharpDX.Direct2D1.SolidColorBrush brush = new SharpDX.Direct2D1.SolidColorBrush(RenderTarget, SharpDX.Color.Blue))     {         RenderTarget.FillEllipse(dot, brush);     }   } } |      |  | | --- | | **Tip**:  One of the advantages of using a Draw.Method is the returned Draw Objects contains metadata which could be used later (such as for obtain the bar index or price value of the dot later on).  If you would use this metadata later on, using a Draw method would be in your best interests.  However, if you are solely looking to render figures on a chart, favoring your custom SharpDX methods can drastically improve performance. |     **Responding to user events**  Do **NOT** use OnRender() for purposes other than rendering.  If you need events to hook into user interactions, consider adding your own event handler.  The example below shows registering the ChartPanel MouseDown event and registering a custom WPF control     |  | | --- | | **Why:**OnRender() may call more or less frequently than you anticipated.  Using your own custom event handlers allows you control and isolate user event logic you are looking to capture |      | ns **Best practice** | | --- | | protected override void OnStateChange() {   if (State == State.Historical)   {     // subscribe to chart panel mouse down event     if (ChartPanel != null) ChartPanel.MouseDown += DoUserClickedChartPanelEvent;       // subscribe to a custom UI element mouse down event     if (myWPFControl != null) myWPFControl.MouseDown += DoCustomWPFControlClickEvent;   }     else if (State == State.Terminated)   {     // remember to unsubscribe when finished     if (ChartPanel != null) ChartPanel.MouseDown -= DoUserClickedChartPanelEvent;     if (myWPFControl != null) myWPFControl.MouseDown -= DoCustomWPFControlClickEvent;   } }   private void DoUserClickedChartPanelEvent(object sender, MouseButtonEventArgs e) {     Print("User clicked on the ChartPanel, executing custom mouse down logic..."); }   private void DoCustomWPFControlClickEvent(object sender, MouseButtonEventArgs e) {     Print("User clicked on my button, executing button logic..."); } |     **Delaying logic for a particular time interval**  Do **NOT** call Thread.Sleep() as it will lock the Instrument thread executing your NinjaScript object.     |  | | --- | | **Why:**Market data events exposed to NinjaScript run on the underlying Instrument thread pool shared by all Instruments. Sleeping the underlying thread of your object will cause the entire Instrument thread to sleep, adversely affecting other features using that same Instrument. |      | ns **Practice to avoid** | | --- | | protected override void OnBarUpdate() {   if (IsFirstTickOfBar && State == State.Realtime)   {     Print("Run some logic before:: " + DateTime.Now);     Thread.Sleep(5000); // sleeping the Instrument thread will have adverse effects on elements outside of your script!     Print("Run some logic after: " + DateTime.Now);   } } |     Instead, try using a Timer object if you need to delay logic execution.     | ns **Best practice** | | --- | | protected override void OnBarUpdate() {   if (IsFirstTickOfBar && State == State.Realtime)   {     // Instead of Thread.Sleep for, create a timer that runs at the desired interval     System.Windows.Forms.Timer timer = new System.Windows.Forms.Timer {Interval = 5000};       // queue the "after" logic to run when the timer elapses     timer.Tick += delegate     {         timer.Stop(); // make sure to stop the timer to only fire ones (if desired)         Print("Run some logic after: " + DateTime.Now);         timer.Dispose(); // make sure to dispose of the timer     };       Print("Run some logic before: " + DateTime.Now);       timer.Start(); // start the timer immediately following the "before" logic   } } | |

[permalink](https://ninjatrader.com/es/support/helpGuides/nt8/index.html?ninjascript_best_practices.htm#Performance)

tog_minus        [Miscellaneous practices](javascript:HMToggle('toggle','MiscellaneousPractices','MiscellaneousPractices_ICON'))

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Floating-point comparison**  Be aware of floating-point precision problems. It can sometimes be more reliable to check within a certain degree of tolerance, such as the [TickSize](https://ninjatrader.com/es/support/helpGuides/nt8/ticksize.htm).     |  | | --- | | **Why**:  You can read more about [Floating-Point Arithmetic](http://ninjatrader.com/support/forum/showthread.php?t=3929) as it applies to NinjaTrader on our support forum |      | ns **Practice to avoid** | | --- | | // depending on how Value[0] was calculated, it could be off by a degree of floating points  // where this logic below would never be true  // e.g., 2050.2499999 vs 2050.50 if (Value[0] == Close[0]) {   // do something } |      | ns**Best practice** | | --- | | // you can avoid these precision issues by rewriting the comparison to evaluate within a certain tolerance. if (Math.Abs(Value[0] - Close[0]) < TickSize) {   // do something }  // You will also see NinjaTrader developed objects use a custom Extension Method // double.ApproxCompare() which Returns an int based on a Epsilon value: if (Close[0].ApproxCompare(Value[0]) == 0) {   // do something } |     **Creating user defined parameter types / enums**  When creating enums for your NinjaScript objects, it is strongly suggested to define those outside the class and in a custom namespace. A reference sample providing all details could be [found here](https://ninjatrader.com/es/support/helpGuides/nt8/creating_a_user-defined_parame.htm).    **Efficiently debugging**  Extremely liberal use of Log() and Print() methods can represent a performance hit on your PC as it takes memory and time to process each one of those method calls. When running custom NinjaScript, especially when using Calculate = Calculate.OnEachTick, please be mindful of how often Log() and Print() methods are processed as it can quickly consume PC resources.    •Log() method should not be used except for critical messages as each log entry makes it to the Control Center log which stays active till the end of the day. Excessive logging can result in huge amounts of memory being allocated just to display all the log messages which would mean less memory for NinjaTrader to do other tasks.  •Print() method can be used more liberally than the Log() method, but can still represent a performance hit if used with extremely high frequency. Consider decreasing the printing from your script if you experience slowdowns when running the script.    **Debug Mode**  The debug mode should only be used if you are actively debugging a script and [attached to a debugger](https://ninjatrader.com/es/support/helpGuides/nt8/visual_studio_debugging.htm).     |  | | --- | | **Why**:  Debug Mode will compile all of the files in the custom project as a "Debug" build, which omits certain optimizations which occur in the C# compilation process.  It is more efficient to use your custom objects in the default "Release" build if you are using your scripts during production. |     **To disable Debug Mode:**  •Right mouse click in any NinjaScript Editor  •Ensure the "Debug Mode" menu item is unchecked  •Press F5 to recompile your scripts  •Your scripts will be re-built using "Release" mode    **Known NinjaScript Wrappers limitations**    •The NinjaScript editor detects code changes in external editors, and will compile on code changes, however code will only be automatically generated by the NinjaScript editor if it's edited within the NinjaScript editor itself (or Visual Studio)  •Wrappers cannot be generated automatically for partial and abstract classes  •Code in the Properties region of the NinjaScript object cannot be commented out with the /\* \*/ style commenting, as it will cause issues with the wrapper generation. Code must be commented out with the // style.  •Subclassing would not allow for wrappers to be generated |

[permalink](https://ninjatrader.com/es/support/helpGuides/nt8/index.html?ninjascript_best_practices.htm" \l "MiscellaneousPractices)

|  |  |
| --- | --- |
| **Navigation:**  [NinjaScript](https://ninjatrader.com/es/support/helpGuides/nt8/ninjascript.htm) > [Educational Resources](https://ninjatrader.com/es/support/helpGuides/nt8/educational_resources.htm) >  **Using SharpDX for Custom Chart Rendering** | [Previous page](https://ninjatrader.com/es/support/helpGuides/nt8/using_images_and_geometry_with_custom_icons.htm) [Return to chapter overview](https://ninjatrader.com/es/support/helpGuides/nt8/educational_resources.htm) [Next page](https://ninjatrader.com/es/support/helpGuides/nt8/working_with_brushes.htm) |

**Understanding the SharpDX .NET Library**

NinjaTrader Chart objects (such as Indicators, Strategies, DrawingTools, ChartStyles) implement an [OnRender()](https://ninjatrader.com/es/support/helpGuides/nt8/onrender.htm) method aimed to render custom lines, shapes, and text to the chart.  To achieve the level of performance required to keep up with market data events, NinjaTrader uses a 3rd-party open-source .NET library named [SharpDX](http://sharpdx.org/).  This 3rd party library provides a C# wrapper for the powerful [Microsoft DirectX API](https://msdn.microsoft.com/en-us/library/windows/desktop/ee663274(v=vs.85).aspx) used for graphics processing and known for its hardware-accelerated performance, including 2D vector and text layout graphics used for **NinjaTrader Chart Rendering**.  The SharpDX/DirectX library is extensive, although NinjaTrader only uses a handful of namespaces and classes, which are documented as a guide in this reference.  In addition to this educational resource, we have also compiled a more focused collection of [SharpDX SDK Reference](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_sdk_reference.htm) resources to help you learn the **SharpDX** concepts used in **NinjaTrader Chart Rendering**.

|  |
| --- |
| **Tips**:  1.There are several pre-installed examples of **OnRender()** and **SharpDX** objects used in the **NinjaTrader.Custom** project.  For starters, please look at the **SampleCustomRender** indicator file  2.Although not entirely identical, the **SharpDX** wrapper is designed to resemble **System.Drawing** namespace; experienced GDI developers will be familiar with concepts discussed in this section.  3.Microsoft provides various [DirectX Programming Guides](https://msdn.microsoft.com/en-us/library/dd372337(v=vs.85).aspx) aimed to educate users with the underlying**C++ DirectX API**.  While **SharpDX (C#)** syntax is different, you may find these guides helpful for understanding **SharpDX** concepts not offered by this guide. |

There are three main **SharpDX** namespaces you need to be familiar with:

|  |  |
| --- | --- |
| [SharpDX](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx.htm) | Contains basic objects used by SharpDX. |
| [SharpDX.Direct2D1](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1.htm) | Contains objects used for rendering for 2D geometry, bitmaps, and text. |
| [SharpDX.DirectWrite](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_directwrite.htm) | Contains objects used for text rendering |

The rest of this page will help you navigate the fundamental concepts needed to achieve custom rendering to your charts.

tog_minus        [SharpDX Vectors and Charting Coordinates](javascript:HMToggle('toggle','SharpDXVectorsAndChartingCoordinates','SharpDXVectorsAndChartingCoordinates_ICON'))

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Understanding the SharpDX.Vector2**  SharpDX Draw methods use a [SharpDX.Vector2](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_vector2.htm) object which describes where to render a command relative to the chart panel.  These **Vector2** objects can be thought as a two-dimensional point in the chart panels X and Y axis. Since the chart canvas used to draw on consists of the full panel of the chart, a vector using a value of 0 for both the X and Y coordinates would  be located in the top left corner of the chart:     | ns | | --- | | // creates a vector located at the top left corner of the chart float x = 0; float y = 0; SharpDX.Vector2 myVector2 = new Vector2(x, y); |      |  | | --- | | **Tip**:   You can learn about [Understanding Chart Canvas Coordinates](https://ninjatrader.com/es/support/helpGuides/nt8/working_with_chart_object_coordinates.htm) on another topic |     **Vector2** objects contain **X**and**Y** properties helpful to recalculate new properties based on the initial vector:     | ns | | --- | | float width = endPoint.X - startPoint.X;  float height = endPoint.Y - startPoint.Y; |     Additionally, you can recalculate a new vector from existing vector objects:     | ns | | --- | | SharpDX.Vector2 center = (startPoint + endPoint) / 2; |     It is also helpful to know that **Vector2** objects are similar to the [Windows Point](https://msdn.microsoft.com/en-us/library/system.windows.point(v=vs.110).aspx) structure and these two types can be used interchangeably.  Depending on the mechanism used to obtain user input or other application values, you may receive the coordinates in a **Point**.  For convenience, NinjaTrader provides a [DXExtension.ToVector2()](https://ninjatrader.com/es/support/helpGuides/nt8/dxextensions_tovector2.htm) method used for converting between these two objects if needed:     | ns | | --- | | SharpDX.Vector2 dxVector2 = wpfPoint.ToVector2(); |     **Calculating Chart Coordinates**  If you simply used a vector with static values, your **Vector2** objects would never change, and your drawing would remain fixed on a particular area of the chart (which may be desired).  However, since NinjaTrader charts are dynamic and responded to various market data updates, scroll, resize, and scale operations - you also need a way to recalculate **vectors** to display information dynamically. To assist in this process, NinjaTrader provides some GUI related utilities to help navigate the chart and calculate values for your custom rendering.     | ns | | --- | | // creates a vector located at the top left corner of the chart panel  startPoint = new SharpDX.Vector2(ChartPanel.X, ChartPanel.Y);    // creates a vector located at the bottom right corner of the chart panel  endPoint = new SharpDX.Vector2(ChartPanel.X + ChartPanel.W, ChartPanel.Y + ChartPanel.H); |     Common utilities fall under 4 key components, and you can learn more about their specific functions from the help guide topics linked in the table below:     |  |  | | --- | --- | | [ChartControl](https://ninjatrader.com/es/support/helpGuides/nt8/chartcontrol.htm) | The entire hosting grid of the Chart | | [ChartBars](https://ninjatrader.com/es/support/helpGuides/nt8/chartbars.htm) | The primary bars series configured on the Chart | | [ChartPanel](https://ninjatrader.com/es/support/helpGuides/nt8/chartpanel.htm) | The panel on which the calling script resides | | [ChartScale](https://ninjatrader.com/es/support/helpGuides/nt8/chartscale.htm) | The Y-Axis values of the configured ChartPanel |      |  | | --- | | **Note**:   For full absolute device coordinates always use **ChartPanel** X, Y, W, H values. **ChartScale** and **ChartControl** properties return WPF units, so they can be drastically different depending on DPI of the user's display.  You can learn about [Working with Pixel Coordinates](https://ninjatrader.com/es/support/helpGuides/nt8/working_with_pixel_coordinates.htm) on another topic. | |

[permalink](https://ninjatrader.com/es/support/helpGuides/nt8/index.html?using_sharpdx_for_custom_chart_rendering.htm#SharpDXVectorsAndChartingCoordinates)

tog_minus        [SharpDX Brush Resources](javascript:HMToggle('toggle','SharpDXBrushResources','SharpDXBrushResources_ICON'))

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Understanding SharpDX Brush Resources**  To color or "paint" an area of the chart, you must define custom resources which describe how you wish the custom render to appear. **SharpDX** contains special resources modeled after the familiar [WPF Brushes](https://ninjatrader.com/es/support/helpGuides/nt8/working_with_brushes.htm). However, the two objects are different in the way they are constructed and also in how they are managed after they are used.    There are many types of **SharpDX Brush Resources** which all derive from the same base [Direct2D1.Brush](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_brush.htm) class.  This base object is not enough to describe how your object should be presented, so in order to use a brush for rendering purposes, you will need to determine exactly what type of brush you wish to use:     |  |  | | --- | --- | | [Direct2D1.SolidColorBrush](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_solidcolorbrush.htm) | Paints an area with a solid color. | | [Direct2D1.RadialGradientBrush](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_radialgradientbrush.htm) | Paints an area with a radial gradient. | | [Direct2D1.LinearGradientBrush](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_lineargradientbrush.htm) | Paints an area with a linear gradient. |     **Describing SolidColorBrush Colors**  The most common and simple brush to use is a [Direct2D1.SolidColorBrush](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_solidcolorbrush.htm)which allows you to paint using a solid color (or with transparency). In the most basic form, **SolidColorBrush** can be constructed using a predefined [SharpDX.Color](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_color.htm)     | ns | | --- | | SharpDX.Direct2D1.SolidColorBrush customDXBrush = new SharpDX.Direct2D1.SolidColorBrush(RenderTarget, SharpDX.Color.DodgerBlue); |     You can also use a [SharpDX.Color3](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_color3.htm) or [SharpDX.Color4](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_color4.htm) structure as a way to get more customizable colors in your rendering:     | ns | | --- | | // create a 3 component color using rgb values in float notation SharpDX.Color3 dxColor3 = new SharpDX.Color3(1.0f, 0.0f, 0.0f);   // create a 4 component color using rgb + alpha (transparency) in float notation SharpDX.Color4 dxColor4 = new SharpDX.Color4(dxColor3, 0.5f);   // solid color brush uses a Color4 during construction SharpDX.Direct2D1.SolidColorBrush argbColorBrush = new SharpDX.Direct2D1.SolidColorBrush(RenderTarget, dxColor4); |     Alternatively, you can set the "transparency" of an existing brush by accessing its [Opacity](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_brush_opacity.htm) property:     | ns | | --- | | customDXBrush.Opacity = .25f; |      |  | | --- | | **Note**:  Unlike their [WPF counterparts](https://ninjatrader.com/es/support/helpGuides/nt8/working_with_brushes.htm), **SharpDX** brushes are thread-safe and do **NOT** need to be frozen. |     **Converting SharpDX Brushes**  **SharpDX Brushes** are **device-dependent resources**, which means they can only be used with the device (i.e., [RenderTarget](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_rendertarget.htm)) which created them.  In practice, this mean you should **ONLY** create your **SharpDX** brushes during the chart object's [OnRender()](https://ninjatrader.com/es/support/helpGuides/nt8/onrender.htm) or [OnRenderTargetChanged()](https://ninjatrader.com/es/support/helpGuides/nt8/onrendertargetchanged.htm) methods.     |  | | --- | | **Warning**:  Failure to create device-dependent resources during the **OnRender()** or **OnRenderTargetChanged()** can lead to a host of issues including memory and application corruption which can negatively impact the stability of NinjaTrader.  Please be careful your **SharpDX** device-dependent resources are only created and updated during either of these two run-time methods.  Please see the [Best Practices for SharpDX Resources](https://ninjatrader.com/es/support/helpGuides/nt8/using_sharpdx_for_custom_chart_rendering.htm#bestpracticesforsharpdxresources) section on this page for more information. |     Because of this detail, a common problem you may run into is the requirement to share a **SharpDX** device brush resource with a **WPF** application brush.  For example, you may have **WPF** brushes defined in the UI during [OnStateChange()](https://ninjatrader.com/es/support/helpGuides/nt8/onstatechange.htm) or recalculated conditionally during [OnBarUpdate()](https://ninjatrader.com/es/support/helpGuides/nt8/onbarupdate.htm), but ultimately wish to use also in custom rendering routines.  For convenience, NinjaTrader provide a [DXExtension.ToDxBrush()](https://ninjatrader.com/es/support/helpGuides/nt8/dxextensions_todxbrush.htm) method used for converting these objects if necessary:     | ns | | --- | | areaBrushDx = areaBrush.ToDxBrush(RenderTarget);  smallAreaBrushDx = smallAreaBrush.ToDxBrush(RenderTarget);  textBrushDx = textBrush.ToDxBrush(RenderTarget); |      |  | | --- | | **Note**: If you are using a large number of brushes, and are not tied to WPF resources, you should favor creating the **SharpDX Brush** directly since the ToDxBrush() method can lead to performance issues if called too frequently during a single render pass.  Please see the [Best Practices for SharpDX Resources](https://ninjatrader.com/es/support/helpGuides/nt8/using_sharpdx_for_custom_chart_rendering.htm#bestpracticesforsharpdxresources) section on this page for more information. | |

[permalink](https://ninjatrader.com/es/support/helpGuides/nt8/index.html?using_sharpdx_for_custom_chart_rendering.htm#SharpDXBrushResources)

tog_minus        [SharpDX RenderTarget](javascript:HMToggle('toggle','SharpDXRenderTarget','SharpDXRenderTarget_ICON'))

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Understanding the RenderTarget**  A [SharpDX Render Target](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_rendertarget.htm) is a general purpose object resource used for receiving and executing drawing commands.  When using a NinjaTrader chart object, a pre-constructed Chart [RenderTarget](https://ninjatrader.com/es/support/helpGuides/nt8/rendertarget.htm) object is available for you to use and ready to receive commands.  You can think of the **RenderTarget** as the device context you are using to render to (i.e. the Chart Panel).  While there is nothing special you need to do to setup this resource, it is important to understand some details regarding the **RenderTarget** to learn how it can be used.    The **RenderTarget** is primarily used for executing commands such as drawing shapes or text:     | ns | | --- | | **RenderTarget**.DrawLine(startPoint, endPoint, areaBrushDx) |     It is commonly used for creating various resources such as **Brushes** and other **SharpDX** objects:     | ns | | --- | | SharpDX.Direct2D1.SolidColorBrush customDXBrush = new SharpDX.Direct2D1.SolidColorBrush(**RenderTarget**, SharpDX.Color.DodgerBlue); |     It can also be used to set various properties to describe how the **RenderTarget** should render:     | ns | | --- | | RenderTarget.AntialiasMode   = SharpDX.Direct2D1.AntialiasMode.PerPrimitive; |     **Sequencing RenderTarget commands**  If the sequence in which objects render is essential to your custom rendering, you will need to be mindful of the order in which you call various **RenderTarget** members. For example, we can draw a second line which uses a different [AntialiasMode](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_antialiasmode.htm) and the renders each line in the order the render target received its commands:     | ns | | --- | | RenderTarget.AntialiasMode = SharpDX.Direct2D1.AntialiasMode.Aliased; RenderTarget.DrawLine(startPoint, endPoint, areaBrushDx, 8);  RenderTarget.AntialiasMode = SharpDX.Direct2D1.AntialiasMode.PerPrimitive; RenderTarget.DrawLine(startPoint, endPoint, customDXBrush, 2); |     In the above example, this order of operations would result in the second [RenderTarget.DrawLine()](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_rendertarget_drawline.htm) to be rendered "on top" of the first **RenderTarget.DrawLine().** If you instead called these two methods in reverse order, you would not see the thinner line since it would be covered up by the thicker line.     |  | | --- | | **Note**:  It is important to realize that **RenderTarget** **sequencing** and the [Chart Object ZOrder](https://ninjatrader.com/es/support/helpGuides/nt8/chart_zorder.htm) are two different concepts. The **ZOrder** property controls the overall layer your entire chart object appears relative to other chart objects existing on the same chart. **RenderTarget sequencing** only affects the order objects are rendered relative itself.  Therefore, it is not possible to sequence your chart object's **RenderTarget** to draw on two different **ZOrders** (e.g., one line above chart bars and another line below). |     **Using the RenderTarget with Device Resources**  Throughout the lifetime of a chart, the render target is created and destroyed several times to satisfy various user commands. As a result, any resources that are created need to be recreated and destroyed as that render target is updated.   The NinjaTrader [OnRenderTargetChange()](https://ninjatrader.com/es/support/helpGuides/nt8/onrendertargetchanged.htm) method was designed to help with this process and will be called anytime the **RenderTarget** has changed.  You should use this method if you have objects which are passed around from various other resources.     |  | | --- | | **Warning**:  Failure to create device-dependent resources during the **OnRender()** or **OnRenderTargetChanged()** can lead to a host of issues including memory and application corruption which can negatively impact the stability of NinjaTrader.  Please be careful your **SharpDX** device-dependent resources are only created and updated during either of these two run-time methods.  Please see the [Best Practices for SharpDX Resources](https://ninjatrader.com/es/support/helpGuides/nt8/using_sharpdx_for_custom_chart_rendering.htm#bestpracticesforsharpdxresources) section on this page for more information. | |

[permalink](https://ninjatrader.com/es/support/helpGuides/nt8/index.html?using_sharpdx_for_custom_chart_rendering.htm#SharpDXRenderTarget)

tog_minus        [SharpDX Lines and Shapes](javascript:HMToggle('toggle','SharpDXLinesAndShapes','SharpDXLinesAndShapes_ICON'))

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **RenderTarget Draw Methods**  All drawings consistent of a few basic shapes which can be called through a handful of **RenderTarget** commands.  "Draw..." methods create just the outline of the shape, and "Fill..." will paint the interior of the shape.     |  |  | | --- | --- | | [RenderTarget.DrawEllipse()](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_rendertarget_drawellipse.htm) | Draws the outline of the specified ellipse using the specified stroke style. | | [RenderTarget.DrawGeometry()](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_rendertarget_drawgeometry.htm) | Draws the outline of the specified geometry using the specified stroke style. | | [RenderTarget.DrawLine()](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_rendertarget_drawline.htm) | Draws a line between the specified points. | | [RenderTarget.DrawRectangle()](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_rendertarget_drawrectangle.htm) | Draws the outline of a rectangle that has the specified dimensions and stroke style. | | [RenderTarget.FillEllipse()](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_rendertarget_fillellipse.htm) | Paints the interior of the specified ellipse. | | [RenderTarget.FillGeometry()](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_rendertarget_fillgeometry.htm) | Paints the interior of the specified geometry. | | [RenderTarget.FillRectangle()](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_rendertarget_fillrectangle.htm) | Paints the interior of the specified rectangle. |      |  | | --- | | **Note**: [AntialiasMode.PerPrimitive](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_antialiasmode.htm) allows for graphics to render more sharply, but comes at a performance cost.  It is recommended to set the [RenderTarget.AntialiasMode](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_rendertarget_antialiasmode.htm) back to the default **AntialiasMode.Aliased** after you finish your **RenderTarget**Draw command.   Please see the [Best Practices for SharpDX Resources](https://ninjatrader.com/es/support/helpGuides/nt8/using_sharpdx_for_custom_chart_rendering.htm#bestpracticesforsharpdxresources) section on this page for more information. |     **Line**  The simplest shape is a Line, executed by the [RenderTarget.DrawLine()](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_rendertarget_drawline.htm) command which just takes two [Vector2](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_vector2.htm) objects which describe where to draw the line, and (optionally) the width of the line to draw:     | ns | | --- | | // create two vectors for the line to draw  SharpDX.Vector2 startPoint = new SharpDX.Vector2(ChartPanel.X, ChartPanel.Y); SharpDX.Vector2 endPoint = new SharpDX.Vector2(ChartPanel.X + ChartPanel.W, ChartPanel.Y + ChartPanel.H);  // define the brush used in the line SharpDX.Direct2D1.SolidColorBrush customDXBrush = new SharpDX.Direct2D1.SolidColorBrush(RenderTarget, SharpDX.Color.DodgerBlue);  // execute the render target draw line with desired values RenderTarget.DrawLine(startPoint, endPoint, customDXBrush, 2);    // always dispose of a brush when finished  customDXBrush.Dispose(); | | render_target_drawline |     **Rectangle**  Using either the [RenderTarget.FillRectangle()](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_rendertarget_fillrectangle.htm) or [RenderTarget.DrawRectangle()](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_rendertarget_drawrectangle.htm) requires a [SharpDX.RectangleF](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_rectanglef.htm) structure, constructed using four values to represent the location (x, y) and size (width, height) of the rectangle to draw.     | ns | | --- | | // create two vectors to position the rectangle  SharpDX.Vector2 startPoint = new SharpDX.Vector2(ChartPanel.X, ChartPanel.Y); SharpDX.Vector2 endPoint = new SharpDX.Vector2(ChartPanel.X + ChartPanel.W, ChartPanel.Y + ChartPanel.H);  // calculate the desired width and heigh of the rectangle float width = endPoint.X - startPoint.X; float height = endPoint.Y - startPoint.Y;    // define the brush used in the rectangle  SharpDX.Direct2D1.SolidColorBrush customDXBrush = new SharpDX.Direct2D1.SolidColorBrush(RenderTarget, SharpDX.Color.DodgerBlue);    // construct the rectangleF struct to describe the with position and size the drawing SharpDX.RectangleF rect = new SharpDX.RectangleF(startPoint.X, startPoint.Y, width, height);  // execute the render target fill rectangle with desired values RenderTarget.FillRectangle(rect, customDXBrush);    // always dispose of a brush when finished  customDXBrush.Dispose(); | | render_target_drawrectangle |     **Ellipse**  Similar to the **Rectangle**, you can draw an **Ellipse** (or circle) using either the [RenderTarget.FillEllipse()](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_rendertarget_fillellipse.htm) or [RenderTarget.DrawEllipse()](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_rendertarget_drawellipse.htm) methods using a [SharpDX Direct2D1 Ellipse](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_ellipse.htm) struct.  For this structure, you will need to use a [Vector2](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_vector2.htm) object to determine the **Center** position of the ellipse, a **RadiusX,** and a **RadiusY** which determines the size of the ellipse:     | ns | | --- | | // create two vectors to position the ellipse  SharpDX.Vector2 startPoint = new SharpDX.Vector2(ChartPanel.X, ChartPanel.Y); SharpDX.Vector2 endPoint = new SharpDX.Vector2(ChartPanel.X + ChartPanel.W, ChartPanel.Y + ChartPanel.H);  // calculate the center point of the ellipse from start/end points  SharpDX.Vector2 centerPoint = (startPoint + endPoint) / 2;    // set the radius of the ellipse float radiusX = 50; float radiusY = 50;    // construct the rectangleF struct to describe the position and size the drawing  SharpDX.Direct2D1.Ellipse ellipse = new SharpDX.Direct2D1.Ellipse(centerPoint, radiusX, radiusY);    // define the brush used in the rectangle  SharpDX.Direct2D1.SolidColorBrush customDXBrush = new SharpDX.Direct2D1.SolidColorBrush(RenderTarget, SharpDX.Color.DodgerBlue);  // execute the render target fill ellipse with desired values RenderTarget.FillEllipse(ellipse, customDXBrush);    // always dispose of a brush when finished  customDXBrush.Dispose(); | | render_target_drawellipse |     **Geometry**  For more complicated shapes, you can use the [RenderTarget.FillGeometry()](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_rendertarget_fillgeometry.htm) or [RenderTarget.DrawGeometry()](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_rendertarget_drawgeometry.htm) methods using a [Direct2D1.PathGeometry](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_pathgeometry.htm) object, which is ultimately defined by a [Direct2D1.GeometrySink](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_geometrysink.htm) interface.     |  | | --- | | **Warning**:  Any **SharpDX PathGeometry** object used in your development must be disposed of after they have been used. NinjaTrader is **NOT** guaranteed to dispose of these resources for you!   Please see the [Best Practices for SharpDX Resources](https://ninjatrader.com/es/support/helpGuides/nt8/using_sharpdx_for_custom_chart_rendering.htm#bestpracticesforsharpdxresources) section on this page for more information. |     To describe a **PathGeometry** object's path, use the object's [PathGeometry.Open()](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_pathgeometry_open.htm) method to retrieve an **GeometrySink**.  Then, use the **GeometrySink** to populate the geometry with figures and segments.  To create a figure, call the [GeometrySink.BeginFigure()](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_geometrysink_beginfigure.htm) method, specify the figure's start point, and then use its Add methods (such as [GeometrySink.AddLine()](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_geometrysink_addline.htm)) to add segments.  When you are finished adding segments, call the [GeometrySink.EndFigure()](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_geometrysink_endfigure.htm) method. You can repeat this sequence to create additional figures. When you are finished creating figures, call the [GeometrySink.Close()](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_geometrysink_close.htm) method.     | ns | | --- | | // create three vectors to position the geometry  SharpDX.Vector2 startPoint = new SharpDX.Vector2(ChartPanel.X, ChartPanel.Y); SharpDX.Vector2 endPoint = new SharpDX.Vector2(ChartPanel.X + ChartPanel.W, ChartPanel.Y + ChartPanel.H); SharpDX.Vector2 centerPoint = (startPoint + endPoint) / 2;   // create the PathGeometry used by the RenderTarget Fill/Draw method SharpDX.Direct2D1.PathGeometry trianglePathGeometry   = new SharpDX.Direct2D1.PathGeometry(Core.Globals.D2DFactory);   // retrieve the GeometrySink used to describe the PathGeometry SharpDX.Direct2D1.GeometrySink geometrySink   = trianglePathGeometry.Open();   // create the points used to define the GeometrySink SharpDX.Vector2 beginPoint = new SharpDX.Vector2(centerPoint.X, startPoint.Y);    // Create a figure using the beginPoint geometrySink.BeginFigure(beginPoint, SharpDX.Direct2D1.FigureBegin.Filled);  // add lines to the figure SharpDX.Vector2 line1 = new SharpDX.Vector2(endPoint.X, centerPoint.Y); geometrySink.AddLine(line1); SharpDX.Vector2 line2 = new SharpDX.Vector2(centerPoint.X, endPoint.Y); geometrySink.AddLine(line2);   // end and close figure when finished geometrySink.EndFigure(SharpDX.Direct2D1.FigureEnd.Closed); geometrySink.Close();  // define the brush used in the geometry  SharpDX.Direct2D1.SolidColorBrush customDXBrush = new SharpDX.Direct2D1.SolidColorBrush(RenderTarget, SharpDX.Color.DodgerBlue);    // execute the render target fill geometry with desired values RenderTarget.FillGeometry(trianglePathGeometry, customDXBrush);    // always dispose of a PathGeometry when finished  trianglePathGeometry.Dispose();  // always dispose of a brush when finished  customDXBrush.Dispose(); | | render_target_fillgeometry |      |  | | --- | | **Tip**:  For more examples of using **Shapes** for custom rendering, many of the DrawingTools included in the **NinjaTrader.Custom** project use these types of **SharpDX** objects and methods extensively. | |

[permalink](https://ninjatrader.com/es/support/helpGuides/nt8/index.html?using_sharpdx_for_custom_chart_rendering.htm#SharpDXLinesAndShapes)

tog_minus        [SharpDX Text Rendering](javascript:HMToggle('toggle','SharpDXTextRendering','SharpDXTextRendering_ICON'))

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Using SharpDX for rendering Text**  Up until this point, we have been using the [SharpDX.Direct2D1](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1.htm) namespace to render shapes.  When dealing with text, there is a separate [SharpDX.DirectWrite](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_directwrite.htm) namespace which works along with the **Direct2D1** objects.    There are two principle objects used for text rendering:  A **TextFormat** object which sets the style of the text, and a **TextLayout** object used to construct complex texts with various settings and provides metrics for measuring the shape the formatted text.    Each one of these objects has their own **RenderTarget** methods: [RenderTarget.DrawText()](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_rendertarget_drawtext.htm) for simple **TextFormat** objects and [RenderTarget.DrawTextLayout()](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_rendertarget_drawtextlayout.htm) for more advanced layouts.  Both methods accept a **TextFormat** object; **DrawTextLayout** is more complicated but has better performance since it reuses the same text layout which does not need to be recalculated.     |  | | --- | | **Tip**:  Both the **TextFormat** and **TextLayout** objects require a **DirectWrite** factory during construction.  For convenience, you can simply use the pre-built NinjaTrader[.Core.Globals.DirectWriteFactory](https://ninjatrader.com/es/support/helpGuides/nt8/directwritefactory.htm) property. |     **Formatting Text**  The **TextFormat** object determines the font size, style and family, among other properties.     |  | | --- | | **Warning**:  Any **SharpDX TextFormat** object used in your development must be disposed of after they have been used. NinjaTrader is **NOT** guaranteed to dispose of these resources for you!  Please see the [Best Practices for SharpDX Resources](https://ninjatrader.com/es/support/helpGuides/nt8/using_sharpdx_for_custom_chart_rendering.htm#bestpracticesforsharpdxresources) section on this page for more information. |      | ns | | --- | | SharpDX.DirectWrite.TextFormat textFormat = new SharpDX.DirectWrite.TextFormat(Core.Globals.DirectWriteFactory, "Arial", 12); |     Once the text formatting has been described, you can use this object to immediately start rendering text in the DrawText() method.  This approach also requires a [SharpDX.RectangleF](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_rectanglef.htm) to help determine the size and position the text renders on the chart.     | ns | | --- | | // define the point for the text to render  SharpDX.Vector2 startPoint = new SharpDX.Vector2(ChartPanel.X, ChartPanel.Y);  // construct the text format with desired font family and size SharpDX.DirectWrite.TextFormat textFormat = new SharpDX.DirectWrite.TextFormat(Core.Globals.DirectWriteFactory, "Arial", 36);  // construct the rectangleF struct to describe the position and size the text SharpDX.RectangleF rectangleF = new SharpDX.RectangleF(startPoint.X, startPoint.Y, ChartPanel.W, ChartPanel.H);    // define the brush used for the text  SharpDX.Direct2D1.SolidColorBrush customDXBrush = new SharpDX.Direct2D1.SolidColorBrush(RenderTarget, SharpDX.Color.DodgerBlue);  // execute the render target text command with desired values RenderTarget.DrawText("I am some text", textFormat, rectangleF, customDXBrush);  // always dispose of textFormat when finished textFormat.Dispose();  // always dipose of brush when finished customDXBrush.Dispose(); | | render_target_drawtext |     **Converting Text**  One common approach to text formatting is to use the same formats as existing chart objects.  This provides familiar text format matching other objects which exist on the chart.  To accomplish this, you can simply use the **ChartControl** NinjaTrader[.Gui.SimpleFont](https://ninjatrader.com/es/support/helpGuides/nt8/simplefont_class.htm) object and convert to **SharpDX** using the [ToDirectWriteTextFormat()](https://ninjatrader.com/es/support/helpGuides/nt8/simplefont_todirectwritetextformat.htm) method.     | ns | | --- | | SharpDX.DirectWrite.TextFormat textFormat = ChartControl.Properties.LabelFont.ToDirectWriteTextFormat(); |     **Text Layouts**  The **TextLayout** object works in combination with the **TextFormat** object by extending its functionality and providing an interface more powerful than a simple Rectangle, enabling you to position, measure, or clip the text to a surrounding shape.    When constructing the **TextLayout** object, you will pass in the exact text as a string you wish to render, along with the desired **TextFormat**.  This gives you the ability to measure the text string after it has been formatted.  During construction, you also have an opportunity to specify the maximum height and width of the **TextLayout**.  For example, we can set the text layout to bound to height and width chart panel:     | ns | | --- | | SharpDX.DirectWrite.TextLayout textLayout = new SharpDX.DirectWrite.TextLayout(Core.Globals.DirectWriteFactory, "I am also some text", textFormat, ChartPanel.W, ChartPanel.H); |     After the text has its format and layout,  you can use the [RenderTarget.DrawTextLayout()](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_rendertarget_drawtextlayout.htm) method to specify the exact location as a [Vector2](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_vector2.htm), as well as the [Brush](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_brush.htm) used to draw the text.   | ns | | --- | | RenderTarget.DrawTextLayout(startPoint, textLayout, customDXBrush); |     **Measuring Text Layouts**  Working with an existing **TextLayout** object, you can use its [TextLayout.Metrics](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_directwrite_textlayout_metrics.htm) object to retrieve metadata related to the size of the formatted text.   This is helpful if you are unsure of the size of the text before it is rendered.  For example, you may wish to draw a rectangle around the formatted text calculated width and height.  Using the approach below, the rectangle will dynamically resize to fit the text values used:     | ns | | --- | | // define the point for the text to render  SharpDX.Vector2 startPoint = new SharpDX.Vector2(ChartPanel.X + 20, ChartPanel.Y + 20);  // construct the text format with desired font family and size SharpDX.DirectWrite.TextFormat textFormat = new SharpDX.DirectWrite.TextFormat(Core.Globals.DirectWriteFactory, "Arial", 36);  // construct the text layout with desired text, text format, max width and height SharpDX.DirectWrite.TextLayout textLayout = new SharpDX.DirectWrite.TextLayout(Core.Globals.DirectWriteFactory, "I am also some text", textFormat, ChartPanel.W, ChartPanel.H);  // create a rectangle which will automatically resize to the width/height of the textLayout SharpDX.RectangleF rectangleF = new SharpDX.RectangleF(startPoint.X, startPoint.Y, textLayout.Metrics.Width, textLayout.Metrics.Height);    // define the brush used for the text and rectangle SharpDX.Direct2D1.SolidColorBrush customDXBrush = new SharpDX.Direct2D1.SolidColorBrush(RenderTarget, SharpDX.Color.DodgerBlue);   // execute the render target draw rectangle with desired values RenderTarget.DrawRectangle(rectangleF, customDXBrush);    // execute the render target text layout command with desired values RenderTarget.DrawTextLayout(startPoint, textLayout, customDXBrush);  // always dispose of textLayout, textFormat, or brush when finished textLayout.Dispose(); textFormat.Dispose(); customDXBrush.Dispose(); | | render_target_drawtextlayout |      |  | | --- | | **Note**:  The **TextLayout.Metrics** height and width properties return the text pixel height, including the line spacing of the font.  Due to the nature of most font families, there will be an amount of line spacing above and below the text.  You can use the [TextLayout.GetLineMetrics()](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_directwrite_textlayout_getlinemetrics.htm) method to help calculate the distance from the top of the text line to its baseline. | |

[permalink](https://ninjatrader.com/es/support/helpGuides/nt8/index.html?using_sharpdx_for_custom_chart_rendering.htm" \l "SharpDXTextRendering)

tog_minus        [SharpDX Stroke Style](javascript:HMToggle('toggle','SharpDXStrokeStyle','SharpDXStrokeStyle_ICON'))

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Using the StrokeStyle Object**  When rendering **SharpDX** Lines and Shapes, you can optionally configure a [SharpDX.Direct2D1.StrokeStyle](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_strokestyle.htm) allowing you to utilize several pre-made [dash styles](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_strokestyle_dashstyle.htm), or even create a custom dash pattern.     |  | | --- | | **Note**:  Unlike other **SharpDX** objects such as **brushes**, the **StrokeStyle** is a device-independent resource.  This means you only need to create the object once throughout the lifetime of the script.  However, the **StrokeStyle** needs to be disposed of when the script is terminated.  The **Creating a Custom DashStyle** example below shows how to use a stroke style from the beginning to end of the lifetime of your script.   Please see the [Best Practices for SharpDX Resources](https://ninjatrader.com/es/support/helpGuides/nt8/using_sharpdx_for_custom_chart_rendering.htm" \l "bestpracticesforsharpdxresources) section on this page for more information. |     For convenience, **SharpDX** provides the [StrokeStyleProperties](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_strokestyleproperties.htm) struct for creating new a **StrokeStyle:**     | ns | | --- | | // create a stroke style property using a pre-configured "DashDot" dash style SharpDX.Direct2D1.StrokeStyleProperties dxStrokeStyleProperties = new SharpDX.Direct2D1.StrokeStyleProperties {   DashStyle = SharpDX.Direct2D1.DashStyle.DashDot, }; |     Once you have your desired stroke style properties, you can create a new stroke style object.     |  | | --- | | **Warning**:  Any **SharpDX StrokeStyle** object used in your development must be disposed of after they have been used. NinjaTrader is **NOT** guaranteed to dispose of these resources for you!   Please see the [Best Practices for SharpDX Resources](https://ninjatrader.com/es/support/helpGuides/nt8/using_sharpdx_for_custom_chart_rendering.htm#bestpracticesforsharpdxresources) section on this page for more information. |      | ns | | --- | | SharpDX.Direct2D1.StrokeStyle dxStrokeStyle = new SharpDX.Direct2D1.StrokeStyle(NinjaTrader.Core.Globals.D2DFactory, dxStrokeStyleProperties); |      |  | | --- | | **Tip**:  The **SharpDX.Direct2D1.StrokeStyle** require a **Direct2D1** factory during construction.  For convenience, you can simply use the pre-built NinjaTrader[.Core.Globals.D2DFactory](https://ninjatrader.com/es/support/helpGuides/nt8/d2dfactory.htm) property. |     And then use that object with the RenderTarget.DrawLine() method:   | ns | | --- | | RenderTarget.DrawLine(startPoint, endPoint, dxBrush, width, dxStrokeStyle); |     **Creating a Custom DashStyle**  By setting the [StrokeStyle.DashStyle](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_strokestyle_dashstyle.htm) property to "**Custom**", you can further refine the appearance of a **SharpDX** rendered line or shape by describing the length and space between the lines. Creating a custom **DashStyle** is not only useful for using **RenderTarget methods**, but also can be used for customizing the appearance of standard [NinjaScript Plots](https://ninjatrader.com/es/support/helpGuides/nt8/addplot.htm).    The code example creates a single **StrokeStyle** object using custom dash style properties.  The example then uses those the custom stroke style object with user defined dashes for overriding the default NinjaTrader plot appearances, and using the same stroke style in a [RenderTarget.DrawLine()](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_rendertarget_drawline.htm) command.     | ns | | --- | | // a SharpDX.Direct2D1.StrokeStyle is device independent // it only needs to be setup once throughout the lifetime of your script private SharpDX.Direct2D1.StrokeStyle dxStrokeStyle;   protected override void OnStateChange() {   if (State == State.SetDefaults)   {     Name = "Custom StrokeStyle";       AddPlot(Brushes.Blue, "Custom StrokeStyle");   }   else if (State == State.Configure)   {     // create a custom stroke style when configured     SharpDX.Direct2D1.StrokeStyleProperties dxStrokeStyleProperties = new SharpDX.Direct2D1.StrokeStyleProperties     {         // set the dash style to "Custom" define the dash pattern         DashStyle = SharpDX.Direct2D1.DashStyle.Custom,           // set further custom/optional StrokeStyle appearances         DashCap = CapStyle.Round,         EndCap   = CapStyle.Flat,         StartCap = CapStyle.Square,         LineJoin = LineJoin.Miter,           // offset in the dash sequence         DashOffset = 10.0f,     };       // define the an array of floating-point values     float[] dashes = { 1.0f, 2.0f, 2.0f, 3.0f, 2.0f, 2.0f };       // create the stroke style using the custom properties and dash array     dxStrokeStyle = new SharpDX.Direct2D1.StrokeStyle(NinjaTrader.Core.Globals.D2DFactory,             dxStrokeStyleProperties, dashes);   }   else if (State == State.Terminated)   {     // make sure to dispose of stroke style when finished     if (dxStrokeStyle != null)     {         if (!dxStrokeStyle.IsDisposed)           dxStrokeStyle.Dispose();     }   } }   protected override void OnBarUpdate() {   Value[0] = Close[0]; }   protected override void OnRender(ChartControl chartControl, ChartScale chartScale) {   // override the appearance of the default plot style   Plots[0].StrokeStyle = dxStrokeStyle;   base.OnRender(chartControl, chartScale);     // use the custom dash style in a RenderTarget.DrawLine() commands   using ( SharpDX.Direct2D1.SolidColorBrush dxBrush = new SharpDX.Direct2D1.SolidColorBrush(RenderTarget, SharpDX.Color.Blue))   {     RenderTarget.DrawLine(new SharpDX.Vector2(ChartPanel.X, ChartPanel.Y), new SharpDX.Vector2(ChartPanel.X + ChartPanel.W, ChartPanel.Y + ChartPanel.H), dxBrush, 2, dxStrokeStyle);   } } | | SharpDX_StrokeStyle | |

[permalink](https://ninjatrader.com/es/support/helpGuides/nt8/index.html?using_sharpdx_for_custom_chart_rendering.htm" \l "SharpDXStrokeStyle)

tog_minus        [Best Practices for SharpDX Resources](javascript:HMToggle('toggle','BestPracticesforSharpDXResources','BestPracticesforSharpDXResources_ICON'))

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Understanding Device-dependent vs Device-independent resources**  Direct2D has several types of resources which may be mapped to the different hardware devices:    •**Device-independent** resources are on the CPU  •**Device-dependent** resources are on the GPU    When **device-dependent** resources are created, system resources are dedicated to that object.  Resources which are **device-dependent** are associated with a particular **RenderTarget** device and are only available on that device.  Therefore, objects which were created using a **RenderTarget** can only be used by that device.  As the **RenderTarget** updates, objects which were previously created will no longer be compatible and can lead to errors.  You can use the NinjaTrader [OnRenderTargetChange()](https://ninjatrader.com/es/support/helpGuides/nt8/onrendertargetchanged.htm)method to detect when the render target has updated and gives you an opportunity to recreate resources.    **Device-dependent resources**  The following objects are associated with a specific **RenderTarget**.  They must be created and dispose of any time the **RenderTarget** is updated:    •[Brush](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_brush.htm)  •[GeometrySink](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_geometrysink.htm)  •[GradientStopCollection](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_gradientstopcollection.htm)  •[LinearGradientBrush](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_lineargradientbrush.htm)  •[RadialGradientBrush](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_radialgradientbrush.htm)  •[SolidColorBrush](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_solidcolorbrush.htm)    **Device-independent resources**  The following objects are **NOT** associated with a specific device.  They can be created once and last for the lifetime of your script, or until they need to be modified:    •[PathGeometry](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_pathgeometry.htm)  •[StrokeStyle](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_strokestyle.htm)  •[TextFormat](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_directwrite_textformat.htm)  •[TextLayout](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_directwrite_textlayout.htm)     |  | | --- | | **Note**:  For more technical information on device resources, please see the [MSDN Direct2D Resources Overview](https://msdn.microsoft.com/en-us/library/dd756757(v=vs.85).aspx) |     **SharpDX DisposeBase**  Although most C# objects stored in memory are handled by the operating system, there are a few **SharpDX** resources which are not managed.  It is important to take care of these resources during the lifetime of your script as there is no guarantee that NinjaTrader will be able to dispose of these unmanaged references for you.    The following commonly used objects implement from the [SharpDX.DisposeBase](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_disposebase.htm) and should be disposed any time they are created:    •[Brush](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_brush.htm)  •[GeometrySink](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_geometrysink.htm)  •[GradientStopCollection](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_gradientstopcollection.htm)  •[LinearGradientBrush](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_lineargradientbrush.htm)  •[PathGeometry](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_pathgeometry.htm)  •[RadialGradientBrush](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_radialgradientbrush.htm)  •[SolidColorBrush](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_solidcolorbrush.htm)  •[StrokeStyle](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_strokestyle.htm)  •[TextFormat](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_directwrite_textformat.htm)  •[TextLayout](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_directwrite_textlayout.htm)     |  | | --- | | **Warning**:  The list above is **NOT** exhaustive and there are other less common **SharpDX** objects that could implement **DisposeBase**. Failure to clean up these resources **WILL** result in NinjaTrader using more memory than necessary and may expose potential "memory leaks" coming from your script.  If you experience unusual amounts of memory being utilized over time, an unmanaged **SharpDX** resource is often times the culprit. |     Since there is no guarantee that NinjaTrader will release objects from memory when your script is terminated, it is best to protect these resources from issues and call [Dispose()](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_disposebase_dispose.htm) as soon as possible.  This commonly involves calling **Dispose(**) at the end of [OnRender()](https://ninjatrader.com/es/support/helpGuides/nt8/onrender.htm),or during [OnRenderTargetChanged()](https://ninjatrader.com/es/support/helpGuides/nt8/onrendertargetchanged.htm) when dealing with **device- dependent** resources such as brush. **Device-independent** resources can be created once and then retained for the life of your application.     | ns | | --- | | protected override void OnRender(ChartControl chartControl, ChartScale chartScale) {   // 1 - setup your resource   SharpDX.Direct2D1.SolidColorBrush customDXBrush = new SharpDX.Direct2D1.SolidColorBrush(RenderTarget, SharpDX.Color.DodgerBlue   // 2 - use your resource   RenderTarget.DrawLine(startPoint, endPoint, customDXBrush);         // 3- dispose of your resource   customDXBrush.Dispose() } |      |  | | --- | | **Note**:  If your resource is setup (i.e., uses the "new" keyword) during **OnRender()** or **OnRenderTargetChange()**, calling **.Dispose()** during [State.Terminated](https://ninjatrader.com/es/support/helpGuides/nt8/state.htm) will **ONLY** dispose of the *very last reference in memory* and is **NOT** sufficient to completely manage all instances created during the lifetime of your script.  You should be diligent in calling **Dispose()**throughout the lifetime of the script. |     You can also consider implementing the [using Statement (C# Reference)](https://msdn.microsoft.com/en-us/library/yh598w02.aspx) which will implicitly call **Dispose() for** you when you are done:     | ns | | --- | | // customDXBrush implicitly calls Dispose() after this block executes  using (SharpDX.Direct2D1.SolidColorBrush customDXBrush = new SharpDX.Direct2D1.SolidColorBrush(RenderTarget, SharpDX.Color.DodgerBlue)) {   RenderTarget.DrawLine(startPoint, endPoint, customDXBrush); } |      |  | | --- | | **Critical**:  Attempting to use an object which has already been disposed can lead to memory corruption that NinjaTrader may not be able to recover.  Attempts to use an object in this manner can result in an error similar to: **Error on calling 'OnRender' method on bar 0: Attempted to read or write protected memory. This is often an indication that other memory is corrupt.** |       You can check to see if can object has been disposed of by using the [DisposeBase.IsDiposed](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_disposebase_isdisposed.htm) property:     | ns | | --- | | SharpDX.Direct2D1.Brush customDXBrush = new SharpDX.Direct2D1.SolidColorBrush(RenderTarget, SharpDX.Color.DodgerBlue);  // checks the object is not disposed of before using if(!customDXBrush.IsDisposed) {   RenderTarget.DrawLine(startPoint, endPoint, customDXBrush);   customDXBrush.Dispose(); } |     You should also favor managing these resources yourself, which means methods which accept a **SharpDX DisposeBase** object as an argument should be created before they are passed into the method and disposed of after they are used.  For example, the code below should be avoided:     | ns | **Practice to avoid** | | --- | --- | | // do NOT convert an object as it is passed to an argument.  // You may have no chance to Dispose of the object!  // Finalizer is not guaranteed to release of these resources RenderTarget.DrawLine(startPoint, endPoint, Brushes.AliceBlue.ToDxBrush(RenderTarget));    MyCustomMethod(Brushes.AliceBlue.ToDxBrush(RenderTarget)); | |     Instead, you should manage these objects yourself:   | ns **Best practice** | | --- | | // Do create and store this reference yourself so you can control when it is released (Y) SharpDX.Direct2D1.Brush customDXBrush = WPFBrush.ToDxBrush(RenderTarget);    RenderTarget.DrawLine(startPoint, endPoint, customDXBrush));    MyCustomMethod(customDXBrush);    customDXBrush.Dipose() |     **Other Best Practices**    If possible, you should avoid using the [ToDxBrush()](https://ninjatrader.com/es/support/helpGuides/nt8/dxextensions_todxbrush.htm) method if it is not necessary.  It is relatively harmless to use this approach for a few brushes, but can introduce performance issues if used too liberally.     | ns **Practice to avoid** | | --- | | // do NOT convert from WPF brushes unnecessarily  SharpDX.Direct2D1.Brush dxBrush1 = System.Windows.Media.Brushes.Blue.ToDxBrush(RenderTarget); SharpDX.Direct2D1.Brush dxBrush2 = System.Windows.Media.Brushes.Red.ToDxBrush(RenderTarget); SharpDX.Direct2D1.Brush dxBrush3 = System.Windows.Media.Brushes.Green.ToDxBrush(RenderTarget); SharpDX.Direct2D1.Brush dxBrush4 = System.Windows.Media.Brushes.Purple.ToDxBrush(RenderTarget); SharpDX.Direct2D1.Brush dxBrush5 = System.Windows.Media.Brushes.Orange.ToDxBrush(RenderTarget); SharpDX.Direct2D1.Brush dxBrush6 = System.Windows.Media.Brushes.Yellow.ToDxBrush(RenderTarget); |     Instead, you should construct a SharpDX Brush directly if a WPF brush is not ever needed:   | ns **Best practice** | | --- | | // Do create SharpDX Brushes directly if you have a large amount of brushes SharpDX.Direct2D1.Brush dxBrush1 = new SharpDX.Direct2D1.SolidColorBrush(RenderTarget, SharpDX.Color.Blue); SharpDX.Direct2D1.Brush dxBrush2 = new SharpDX.Direct2D1.SolidColorBrush(RenderTarget, SharpDX.Color.Red); SharpDX.Direct2D1.Brush dxBrush3 = new SharpDX.Direct2D1.SolidColorBrush(RenderTarget, SharpDX.Color.Green); SharpDX.Direct2D1.Brush dxBrush4 = new SharpDX.Direct2D1.SolidColorBrush(RenderTarget, SharpDX.Color.Purple); SharpDX.Direct2D1.Brush dxBrush5 = new SharpDX.Direct2D1.SolidColorBrush(RenderTarget, SharpDX.Color.Orange); SharpDX.Direct2D1.Brush dxBrush6 = new SharpDX.Direct2D1.SolidColorBrush(RenderTarget, SharpDX.Color.Yellow); |     Rendering with anti-aliasing disabled can be used to render a higher qualify shapes but comes as a performance impact.  You should make sure to set this render target property back to its default when you are finished with a render routine.     | ns **Best practice** | | --- | | // AntialiasMode.PerPrimitive is more resource intensive  // store the old reference before setting the desired value SharpDX.Direct2D1.AntialiasMode oldAntialiasMode = RenderTarget.AntialiasMode; RenderTarget.AntialiasMode = SharpDX.Direct2D1.AntialiasMode.PerPrimitive;   // execute your render routines   // and then set back to initial AntialiasMode when finished RenderTarget.AntialiasMode = oldAntialiasMode; | |

[permalink](https://ninjatrader.com/es/support/helpGuides/nt8/index.html?using_sharpdx_for_custom_chart_rendering.htm#BestPracticesforSharpDXResources)

|  |  |
| --- | --- |
| **Navigation:**  [NinjaScript](https://ninjatrader.com/es/support/helpGuides/nt8/ninjascript.htm) > [Language Reference](https://ninjatrader.com/es/support/helpGuides/nt8/language_reference_wip.htm) > [Strategy](https://ninjatrader.com/es/support/helpGuides/nt8/strategy.htm) >  **StrategyBaseConverter Class** | [Previous page](https://ninjatrader.com/es/support/helpGuides/nt8/stoptargethandling.htm) [Return to chapter overview](https://ninjatrader.com/es/support/helpGuides/nt8/strategy.htm) [Next page](https://ninjatrader.com/es/support/helpGuides/nt8/systemperformance.htm) |

**Definition**

A custom [TypeConverter](https://msdn.microsoft.com/en-us/library/system.componentmodel.typeconverter%28v=vs.110%29.aspx) class handling the designed behavior of a strategy's property descriptor collection.  Use this as a base class for any custom **TypeConverter** you are applying to a strategy class.

|  |
| --- |
| **Notes:**  •A working NinjaScript demo can be found through the reference sample on "[Using a TypeConverter to Customize Property Grid Behavior](http://ninjatrader.com/support/forum/showthread.php?t=97919)"  •When applying the custom converter, you must fully qualify the name (e.g., "NinjaTrader.NinjaScript.Strategies.MyCustomConveter")  •Additional **TypeConverter** information can be found from the [MSDN documentation](https://msdn.microsoft.com/en-us/library/system.componentmodel.typeconverter%28v=vs.110%29.aspx)  •See also [TypeConverterAttribute](https://ninjatrader.com/es/support/helpGuides/nt8/typeconverterattribute.htm)  •For Indicators, see the [IndicatorBaseConverter](https://ninjatrader.com/es/support/helpGuides/nt8/indicatorbaseconverter.htm) class |

**Relevant base methods**

|  |  |
| --- | --- |
| [TypeConverter.GetProperties()](https://msdn.microsoft.com/en-us/library/system.componentmodel.typeconverter.getproperties(v=vs.110).aspx) | When overriding **GetProperties()**, calling base.GetProperties() ensures that all default property grid behavior works as designed |
| [TypeConverter.GetPropertiesSupported()](https://msdn.microsoft.com/en-us/library/system.componentmodel.typeconverter.getpropertiessupported(v=vs.110).aspx) | In your custom converter class, you must override **GetPropertiesSupported()**and return a value of **true** in order for your custom type converter to work |

**Syntax**

public class StrategyBaseConverter : TypeConverter

|  |
| --- |
| **Warning**:  Failure to apply a type of **StrategyBaseConverter** on an strategy class can result in unpredictable behavior of the standard NinjaTrader WPF property grid. |

|  |
| --- |
| **Tip**: Common strategy functions like Print() are not available to a type converter instance.  To debug a type converter class, you can use the AddOn [Debug Concepts](https://ninjatrader.com/es/support/helpGuides/nt8/alert_and_debug_concepts.htm) or [attach to a debugger](https://ninjatrader.com/es/support/helpGuides/nt8/visual_studio_debugging.htm) (recommended) |

**Examples**

| ns | |
| --- | --- |
| //This namespace holds Strategies in this folder and is required. Do not change it. namespace NinjaTrader.NinjaScript.Strategies {   // When applying the type converter, you must fully qualify the name   [TypeConverter("NinjaTrader.NinjaScript.Strategies.MyCustomConveter")]   public class MyCustomStrategy : Strategy   {     protected override void OnStateChange()     {         if (State == State.SetDefaults)         {           Name                             = "MyCustomStrategy";         }     }       protected override void OnBarUpdate()     {         //Add your custom strategy logic here.     }   }     // custom converter class for strategies   public class MyCustomConveter : StrategyBaseConverter   {     // A general TypeConveter method used for converting types     public override PropertyDescriptorCollection GetProperties(ITypeDescriptorContext context, object component, Attribute[] attrs)     {         // sometimes you may need the strategy instance which actually exists on the grid         MyCustomStrategy strategy = component as MyCustomStrategy;           // base.GetProperties ensures we have all the properties (and associated property grid editors)         // NinjaTrader internal logic handles for a given strategy         PropertyDescriptorCollection propertyDescriptorCollection = base.GetPropertiesSupported(context)                 ? base.GetProperties(context, component, attrs) : TypeDescriptor.GetProperties(component, attrs);           if (strategy == null || propertyDescriptorCollection == null)           return propertyDescriptorCollection;           // example of why you may need the instance that exists on the grid....         if (strategy.EntryHandling == EntryHandling.UniqueEntries)         {           // do something in the event a property contains some value...         }           // Loop all of the properties of the strategy         foreach (PropertyDescriptor property in propertyDescriptorCollection)         {           // do something with a specific property             // cannot call Print() here           // but you can call the static Output window "Process()"           NinjaTrader.Code.Output.Process(property.Name, PrintTo.OutputTab1);         }           // must return the collection after making changes         return propertyDescriptorCollection;     }       // Important:  This must return true otherwise the type converter will not be called     public override bool GetPropertiesSupported(ITypeDescriptorContext context)     { return true; }   } } | |
| **Navigation:**  [NinjaScript](https://ninjatrader.com/es/support/helpGuides/nt8/ninjascript.htm) > [Language Reference](https://ninjatrader.com/es/support/helpGuides/nt8/language_reference_wip.htm) > [Common](https://ninjatrader.com/es/support/helpGuides/nt8/common.htm) > [Attributes](https://ninjatrader.com/es/support/helpGuides/nt8/attributes.htm) >  **TypeConverterAttribute()** | | [Previous page](https://ninjatrader.com/es/support/helpGuides/nt8/rangeattribute.htm) [Return to chapter overview](https://ninjatrader.com/es/support/helpGuides/nt8/attributes.htm) [Next page](https://ninjatrader.com/es/support/helpGuides/nt8/xmlignoreattribute.htm) |

**Definition**

Binds an object or property to a specific **TypeConverter** implementation.  This is commonly used to customize property descriptors on the NinjaTrader property grid.

|  |
| --- |
| **Notes**:  •If you are looking to extend behavior of an Indicator or Strategy (e.g., values of one property influence another), you must implement either an [IndicatorBaseTypeConverter](https://ninjatrader.com/es/support/helpGuides/nt8/indicatorbaseconverter.htm) or [StrategyBaseTypeConveter](https://ninjatrader.com/es/support/helpGuides/nt8/strategybaseconverter.htm).  This is to ensure default property descriptor behavior works as intended.  •For converting types of a specific property, implementing a standard **TypeConverter** is sufficient  •A working NinjaScript demo can be found through the reference sample on "Using a TypeConverter to Customize Property Grid Behavior"  •The **TypeConverterAttribute** object is a general purpose attribute made available from the .NET Framework. The information on this page is written to demonstrate how you may use this object within NinjaScript conventions used with the NinjaTrader UI's property grid (e.g., an indicator dialog).  There are more methods and properties that you can learn about from **MSDN's** [TypeConverterAttribute Class](https://msdn.microsoft.com/en-us/library/system.componentmodel.typeconverterattribute(v=vs.110).aspx) which are**NOT** covered in this topic; as such there is **NO** guarantee they will work with the NinjaTrader UI's property grids. |

**Syntax**

TypeConverterAttribute(string)  
TypeConverterAttribute(type)

**Examples**

| ns | |
| --- | --- |
| // Only applied to this property: can just implement a general TypeConveter [TypeConverter(typeof(MyCustomBoolConveter))] public bool CustomBool    // Applied to the entire indicator: must implement an IndicatorBaseTypeConveter [TypeConverter("NinjaTrader.NinjaScript.Indicators.MyConverter")] public class MyCustomIndicator : Indicator {   }   // Applied to the entire strategy: must implement a StrategyBaseTypeConveter [TypeConverter("NinjaTrader.NinjaScript.Strategies.MyCustomConveter")] public class MyCustomStrategy : Strategy {   } | |
| **Navigation:**  [NinjaScript](https://ninjatrader.com/es/support/helpGuides/nt8/ninjascript.htm) > [Educational Resources](https://ninjatrader.com/es/support/helpGuides/nt8/educational_resources.htm) > [Reference Samples](https://ninjatrader.com/es/support/helpGuides/nt8/reference_samples.htm) > [Indicator](https://ninjatrader.com/es/support/helpGuides/nt8/indicator2.htm) >  **Using a TypeConverter to Customize Property Grid Behavior** | | [Previous page](https://ninjatrader.com/es/support/helpGuides/nt8/using_a_series_or_dataseries_o.htm) [Return to chapter overview](https://ninjatrader.com/es/support/helpGuides/nt8/indicator2.htm) [Next page](https://ninjatrader.com/es/support/helpGuides/nt8/using_custom_events_to_output_.htm) |

The default behavior of the NinjaTrader 8 property grid is designed to handle the most basic display of your custom Indicator and Strategy properties. However, there are special cases where you may want to control how unique properties display to other users. Since using a TypeConverter is more of a general C# concept used to convert values between data types (a string to an enum for example), so the possibilities of what you can do are within the bounds of the .NET Framework. This NinjaScript sample was produced to help NinjaScript developers understand how to leverage the IndicatorBaseConverter and StrategyBaseConverter helper classes to customize property grid behavior without affecting general NinjaTrader 8 property behavior.

**Note**

•NinjaTrader 8 expects custom properties to be of value type or type which implements ICloneable interface

•This reference sample assumes you are familiar with basic C# TypeConverter practices and is intended as a starting point. There are extensive 3rd party guides available online which can help you implement a particular type converter goal not covered in this sample

**Key concepts in this example**

There are 5 use cases demonstrated in this sample which fall into two different categories:

1. Dynamically manipulate what is displayed on the UI Property Grid

a.Show / hide properties based on secondary input

b.Disable / enable properties based on secondary input

2. Customizing how a property is displayed

a.Display "Friendly" enum values

b.Re-implement a "bool" CheckBox as "Friendly" a ComboBox

c.Display a custom collection / list with user defined values at run time

**Important related documentation**

C#

•[ICloneable Interface](https://msdn.microsoft.com/en-us/library/system.icloneable%28v=vs.110%29.aspx)

•[PropertyDescriptor Class](https://msdn.microsoft.com/en-us/library/system.componentmodel.propertydescriptor%28v=vs.110%29.aspx)

•[RefreshPropertiesAttribute](https://msdn.microsoft.com/en-us/library/system.componentmodel.refreshpropertiesattribute(v=vs.110).aspx)

•[TypeConverter Class](https://msdn.microsoft.com/en-us/library/system.componentmodel.typeconverter%28v=vs.110%29.aspx)

•[DisplayAttribute Class](https://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.displayattribute(v=vs.110).aspx)

NinjaTrader

•[IndicatorBaseConverter Class](https://ninjatrader.com/es/support/helpGuides/nt8/indicatorbaseconverter.htm)

•[StrategyBaseConverter Class](https://ninjatrader.com/es/support/helpGuides/nt8/strategybaseconverter.htm)

•[TypeConverterAttribute](https://ninjatrader.com/es/support/helpGuides/nt8/typeconverterattribute.htm)

|  |
| --- |
| **Note**: The reference sample files on this page are written for an indicator using the **IndicatorBaseConverter** class, but the same key concepts are available to strategies by replacing the **StrategyBaseConverter** where noted in the sample code. |

**Import instructions**

1.Download the file contained in this Help Guide topic to your PC desktop

2.From the Control Center window, select the menu Tools > Import > NinjaScript

3.Select the downloaded file

**Additional resources**

MSDN - How to: [Implement a Type Converter](https://msdn.microsoft.com/en-us/library/ayybcxe5.aspx)

[SampleIndicatorTypeConverter\_NT8.zip](https://ninjatrader.com/support/helpGuides/nt8/samples/SampleIndicatorTypeConverter_NT8.zip)

|  |  |
| --- | --- |
| **Navigation:**  [NinjaScript](https://ninjatrader.com/es/support/helpGuides/nt8/ninjascript.htm) > [Educational Resources](https://ninjatrader.com/es/support/helpGuides/nt8/educational_resources.htm) >  **Multi-Time Frame & Instruments** | [Previous page](https://ninjatrader.com/es/support/helpGuides/nt8/multi-threading.htm) [Return to chapter overview](https://ninjatrader.com/es/support/helpGuides/nt8/educational_resources.htm) [Next page](https://ninjatrader.com/es/support/helpGuides/nt8/understanding_the_lifecycle_of.htm) |

**Multi-Series Scripting Overview**

NinjaScript supports multiple time frames and instruments in a single script. This is possible because you can add additional Bars objects to indicators or strategies, in addition to the primary Bars object to which they are applied. A Bars object represents all of the bars of data on a chart. For example, if you had a MSFT 1 minute chart with 200 bars on it, the 200 bars represent one Bars object. In addition to adding Bars objects for reference or for use with indicator methods, you can execute trades across all the different instruments in a script. There is extreme flexibility in the NinjaScript model that NinjaTrader uses for multiple-bars scripts, so it is very important that you understand how it all works before you incorporate additional Bars objects in a script. An important fact to understand is that NinjaScript is truly event driven; every Bars object in a script will call the [OnBarUpdate()](https://ninjatrader.com/es/support/helpGuides/nt8/onbarupdate.htm) method. The significance of this will become evident throughout this page.

|  |
| --- |
| **Note**:  If using [OnMarketData()](https://ninjatrader.com/es/support/helpGuides/nt8/onmarketdata.htm), a subscription will be created on all bars series added in your indicator or strategy strategy (even if the instrument is the same).  The market data subscription behavior occurs both in real-time and during [TickReplay](https://ninjatrader.com/es/support/helpGuides/nt8/developing_for__tick_replay.htm) historical |

It is also important that you understand the following method and properties:

•[AddDataSeries()](https://ninjatrader.com/es/support/helpGuides/nt8/adddataseries.htm)

•[BarsArray](https://ninjatrader.com/es/support/helpGuides/nt8/barsarray.htm)

•[BarsInProgress](https://ninjatrader.com/es/support/helpGuides/nt8/barsinprogress.htm)

•[CurrentBars](https://ninjatrader.com/es/support/helpGuides/nt8/currentbars.htm)

|  |
| --- |
| **Note**:  As we move through this section, the term "Primary Bars" will be used and for the purpose of clarification, this will always refer to the first Bars object loaded into a script. For example, if you apply a script on MSFT 1 minute chart, the primary Bars would be MSFT 1 minute data set.    **This section is written in sequential fashion. Example code is re-used and built upon from sub section to sub section.** |

tog_minus        [Working With Multi-Time Frame Objects](javascript:HMToggle('toggle','WorkingWithMultiTimeFrameObjects','WorkingWithMultiTimeFrameObjects_ICON'))

|  |  |  |
| --- | --- | --- |
| **Data processing sequence**  Understanding the sequence in which bars series process and the granularity provided by market data vendors is essential for efficient multi-series development. Let’s assume we have two series (primary and secondary) in our script, which is representing the same instrument, yet different intervals.  During historical data processing, NinjaTrader updates the two series *strictly* according to their timestamps, calling the primary bar series of the corresponding timestamps first, and then calling the secondary series.     |  | | --- | | **Note**:  Historical bars are processed according to their timestamps with the primary bars first, followed by the secondary, which is **NOT** guaranteed to be the same sequence that these events occurred in real-time.  If your development requires ticks to process in the same sequence historically as well as in real-time, you will need to enable [Tick Replay](https://ninjatrader.com/es/support/helpGuides/nt8/developing_for__tick_replay.htm) (utilizes more PC resources). |       **Shared Timestamps**  In circumstances where multiple bars share the same exact timestamps, your primary bars series will *always* be processed first, followed by the secondary bars series (regardless of the period value used). Consequently, if you were looking to obtain a value from the secondary bars series, it would **ONLY** be available *after* the primary series has been processed for the same timestamps. For example, consider a news event or a fast moving market with an influx of ticks (session begin/session end). This activity will yield a wider range of bars than usual and the probability of those bars sharing the same timestamps increases. If such a succession of bars with the same timestamps is processed, the primary bars would be processed first and then the secondary bars during this period.     |  | | --- | | **Tip**: While the following behavior applies to all period types, the effects are amplified on smaller time frames.  If you plan on using a high-resolution (e.g., 1-second, 10-tick, etc), please make sure to thoroughly read and understand the material below when working with these additional series.  It is also important to keep in mind that the granularity of the timestamps will dictate how accurately NinjaTrader can synchronize the bars in historical processing.   The available level of granularity will be dependent upon which [data provider](https://ninjatrader.com/es/support/helpGuides/nt8/data_by_provider.htm) you use with NinjaTrader. |       Let’s look at an illustration of how the multi-time frame bar processing sequence can be understood.  Assume our primary series is a 5-tick bar series, and our secondary series is a 1-tick bar series.  The time of day is near the session close, so a rapid sequence of bars is generated.    In the figure below, the 1st group of bars (colored orange), and the 4th group of bars (colored purple) process in an exact logical sequence (i.e., a single primary bar update, followed by five secondary series updates).  This is because each bar in these groups have *unique timestamps* and NinjaTrader can synchronize those bars logically in the exact time sequence each series updated.  However, all of the bars marked with red text share the*same exact timestamps* down to the millisecond (14:59:00:480).  Since there were six ticks in sequence with the shared timestamps, this range of ticks expands two of the primary bars (colored green and blue).  As a result, the primary bar #3 appears to update earlier when compared to the secondary series.  In reality, both bars series are incrementing in their exact sequence according to the timestamps of each series.      same_timestamp_bars  Figure 1.  Bar processing  with shared timestamps  1.Timestamps of primary series (hour, minute, second, millisecond)  2.Current bars numbered in series representing 5-tick primary series  3.Current bars numbered in series representing 1-tick secondary series  4.Millisecond time stamps of secondary series  5.A sequence of bars sharing the same time stamps |

[permalink](https://ninjatrader.com/es/support/helpGuides/nt8/index.html?multi-time_frame__instruments.htm#WorkingWithMultiTimeFrameObjects)

tog_minus        [Adding Additional Bars Objects to NinjaScript](javascript:HMToggle('toggle','AddingAdditionalBarsObjectToninjascript','AddingAdditionalBarsObjectToninjascript_ICON'))

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Additional Bars are added to a script via the [AddDataSeries()](https://ninjatrader.com/es/support/helpGuides/nt8/adddataseries.htm) method in the [OnStateChange()](https://ninjatrader.com/es/support/helpGuides/nt8/onstatechange.htm) method when the [State](https://ninjatrader.com/es/support/helpGuides/nt8/state.htm) has reached **State.Configure**. When a Bars object is added to a script, it is also added to the [BarsArray](https://ninjatrader.com/es/support/helpGuides/nt8/barsarray.htm) array. **BarsArray** functions like a container in the script that holds all Bars objects added to the script. As a Bars object is added to the script, it's added to **BarsArray** and given an index number so we can retrieve this Bars object later.     |  | | --- | | **Warning:**  •This method should **ONLY** be called from the [OnStateChange()](https://ninjatrader.com/es/support/helpGuides/nt8/onstatechange.htm)method during **State.Configure**  •Arguments supplied to **AddDataSeries()** should be hardcoded and **NOT** dependent on run-time variables which cannot be reliably obtained during [State.Configure](https://ninjatrader.com/es/support/helpGuides/nt8/state.htm) (e.g., [Instrument](https://ninjatrader.com/es/support/helpGuides/nt8/instrument.htm), [Bars](https://ninjatrader.com/es/support/helpGuides/nt8/bars.htm), or user input).  Attempting to add a data series dynamically is **NOT** guaranteed and therefore should be avoided.  Trying to load bars dynamically may result in an error similar to: **Unable to load bars series. Your NinjaScript may be trying to use an additional data series dynamically in an unsupported manner.**  •When instantiating indicators in a Multi-Series script in [OnStateChange](https://ninjatrader.com/es/support/helpGuides/nt8/onstatechange.htm), the input any hosted indicator is running on should be explicitly stated (since a specific [BarsInProgress](https://ninjatrader.com/es/support/helpGuides/nt8/barsinprogress.htm) is not guaranteed) |     For the purpose of demonstration, let's assume that a MSFT 1 minute bar is our primary Bars object (set when the script is applied to a 1 minute MSFT chart) and that the OnStateChange() method is adding a 3 minute Bars object of MSFT, then adding a 1 minute Bars object of AAPL, for a total of 3 unique Bars objects.     | ns | | --- | | protected override void OnStateChange() {   if (State == State.SetDefaults)   {                     Name   = "Multi-Time Frame & Instruments Example";   }   else if (State == State.Configure)   {       AddDataSeries(BarsPeriodType.Minute, 3);     AddDataSeries("AAPL", BarsPeriodType.Minute, 1);   } } |      |  | | --- | | **Note**: To maximize data loading performance, any NinjaScript object (indicator or strategy as host) which references a multi-series indicator which calls AddDataSeries must include it's own calls to AddDataSeries(). For example, if the code above was included in an indicator, and that indicator was referenced in a  NinjaScript strategy, then the hosting strategy will need to include the same calls to AddDataSeries(). When the strategy adds the additional Bars objects, the calls to AddDataSeries() within the indicator will be ignored. If the Bars objects are not added by the strategy in such cases, and error will be thrown in the Log tab of the Control Center that would read - "A hosted indicator tried to load additional data. All data must first be loaded by the hosting NinjaScript in its configure state." | |

[permalink](https://ninjatrader.com/es/support/helpGuides/nt8/index.html?multi-time_frame__instruments.htm#AddingAdditionalBarsObjectToninjascript)

tog_minus        [Creating Series<T> Objects](javascript:HMToggle('toggle','CreatingSeriesObjects','CreatingSeriesObjects_ICON'))

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Series<T> Objects**  [Series<T>](https://ninjatrader.com/es/support/helpGuides/nt8/seriest.htm) is the base class for [PriceSeries](https://ninjatrader.com/es/support/helpGuides/nt8/priceseries.htm), [TimeSeries](https://ninjatrader.com/es/support/helpGuides/nt8/timeseries.htm), and [VolumeSeries](https://ninjatrader.com/es/support/helpGuides/nt8/volumeseries.htm). Rather than using one of these pre-defined derived classes, you can create your own Series<T> collection to hold any Type that you choose. The advantage that Series<T> has over other collections is that it can be quickly initialized to contain a number of index slots equal to the number of bars in one of the Bars objects on the chart, with each index slot corresponding to a specific bar.    **Initializing a Series<T> with BarsArray**  A Series<T> can be constructed by passing in a specific index of BarsArray. Initializing a Series<T> this way produces an empty Series<T> container holding the same number of index slots as the BarsArray that was passed in as an argument. For example, assuming that BarsArray[1] holds 500 bars, the code below will create an empty Series<T> with 500 index slots:     | ns **Initializing Series<T> with BarsArray** | | --- | | private Series<double> myEmptyIndexedSeries; // Define a Series<T> objectvariable.   // Initialize the Series object to have the same number of index slots as BarsArray[1] protected override void OnStateChange() {   if (State == State.DataLoaded)   {       // Passing in BarsArray[1] as an argument results in an empty Series with an identical number of index slots       myEmptyIndexedSeries = new Series<double>(BarsArray[1]);   } } |       This method of initializing a Series<T> can be especially useful when you wish to store user-defined information related to each bar in a Bars object on the chart. This process ensures that index slots are available for every bar on the chart right away.    **Initializing a Series<T> with an Indicator Method**  Passing in an indicator method as an argument when instantiating a Series<T> object provides an alternative to the process outlined above. Because indicator methods already contain Series objects synced to the bars on a chart, they can be used to inform the constructor of Series<T> of how many index slots to create.     | ns **Initializing Series<T> with an Indicator Method** | | --- | | // Declare two Series objects private Series<double> primarySeries; private Series<double> secondarySeries;   protected override void OnStateChange() {     if (State == State.Configure)   {       // Adds a secondary bar object to the strategy.       AddDataSeries(BarsPeriodType.Minute, 5);   }   else if (State == State.DataLoaded)   {       // Syncs a Series object to the primary bar object       primarySeries = new Series<double>(this);               /\* Syncs another Series object to the secondary bar object.        We use an arbitrary indicator overloaded with an ISeries<double> input to achieve the sync.        The indicator can be any indicator. The Series<double> will be synced to whatever the        BarsArray[] is provided.\*/       secondarySeries = new Series<double>(SMA(BarsArray[1], 50));               // Stop-loss orders are placed 5 ticks below average entry price       SetStopLoss(CalculationMode.Ticks, 5);         // Profit target orders are placed 10 ticks above average entry price       SetProfitTarget(CalculationMode.Ticks, 10);   } } | |

[permalink](https://ninjatrader.com/es/support/helpGuides/nt8/index.html?multi-time_frame__instruments.htm#CreatingSeriesObjects)

tog_minus        [How Bars Data is Referenced](javascript:HMToggle('toggle','HowBarsDataIsReferenced','HowBarsDataIsReferenced_ICON'))

|  |
| --- |
| Understanding how multi-time frame bars are processed and which OHLCV data is referenced is critical.    Figure 1 below demonstrates the concept of bar processing on historical data or in real-time when the [Calculate](https://ninjatrader.com/es/support/helpGuides/nt8/calculate.htm) property is set to**Calculate.OnBarClose**. The 1 minute bars in yellow will only know the OHLCV of the 3 minute bar in yellow. The 1 minute bars in cyan will only know the OHLCV data of the 3 minute bar in cyan. Take a look at "Bar #5," which is the fifth one minute bar. If you wanted to know the current high value for the 3-minute time frame, you would get the value of the first 3 minute bar since this is the last "closed" bar. The second 3 minute bar (cyan) would not be known at that time.    Tips_3  Figure 1.  Bar processing on historical data using Calculate.OnBarClose  1.Primary 1-minute bar series  2.Secondary 3-minute bar series  3.Bar #5    Contrast the above image and concept with the image below, which demonstrates bar processing in real-time when the **Calculate** property is set to **Calculate.OnEachTick**  (tick by tick processing) or **Calculate.OnPriceChange (**processing by change in price). The 1 minute bars in yellow will know the current OHLCV of the 3 minute bar in yellow (second 3 minute bar) which is still in formation and has not yet closed.    Tips_4  Figure 2.  Bar processing in real-time using Calculate.OnEachTick or Calculate.OnPriceChange  1.Primary 1-minute bar series  2.Secondary 3-minute bar series  3.Bar #5    If you have a multi-time frame script in real-time, and it is processing tick by tick instead of on the close of each bar, understand that the OHLCV data you access in real-time is different than on historical data.    Below is another example to illustrate this point:    Your script has complex logic that changes the bar color on the chart. You are running tick by tick, as per the above "Figure 2" image, the 5th 1 minute bar is looking at OHLCV data from the second 3 minute bar. Your script changes the fifth 1 minute bar color to green. In the future, you reload your script into the chart and the fifth 1 minute bar is now a historical bar. As per Figure 1 above, the fifth 1 minute bar now references the OHLCV data of the first 3 minute bar (instead of the 2nd 3 minute bar as per Figure 2) and as a result, your script logic condition for coloring the bar green is no longer valid. The result is that now your chart looks different.    Special considerations for session boundaries :    Bars are not considered closed until the first tick of the following bar comes in (see also "True Event Driven OnBarUpdate" below). As a consequence, if the above series 2 cyan bar represents the final bar of a session, and this bar is referenced from the matching series 1 cyan bar, or anywhere after that, the data from the close of the bar (the beginning of the next session) will be referenced. If you plan on using multiple session templates, you will need to handle the final bar of a trading day case explicitly (for example, using a [Session Iterator](https://ninjatrader.com/es/support/helpGuides/nt8/sessioniterator.htm) and the [PriorDayOHLC](https://ninjatrader.com/es/support/helpGuides/nt8/prior_day_ohlc.htm)) if you would like to reference data from the end of the previous trading day instead of the beginning of the current trading day. |

[permalink](https://ninjatrader.com/es/support/helpGuides/nt8/index.html?multi-time_frame__instruments.htm#HowBarsDataIsReferenced)

tog_minus        [Using Bars Objects as Input to Indicator Methods](javascript:HMToggle('toggle','UsingBarsObjectsAsInputToIndicatorMethods','UsingBarsObjectsAsInputToIndicatorMethods_ICON'))

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| In the sub-section above, the concept of index values was introduced. This is a critical concept to understand since it is used consistently when working with multi-Bars script.    Let's demonstrate this concept:    Carrying on from the example above, our primary Bars is set from a MSFT 1 minute chart    **MSFT 1 minute Bars is given an index value of 0 in BarsArray**    In the OnStateChange() method we added a MSFT 3 minute Bars object and an AAPL 1 minute Bars object to the script    **MSFT 3 minute Bars is given an index value of 1 in BarsArray**  **AAPL 1 minute Bars is given an index value of 2 in BarsArray**    Incremental index values are given to Bars objects as they are added to a script. If there are 10 Bars objects in a script, then you will have index values ranging from 0 to 9.    Our script now has 3 Bars objects in the container **BarsArray**. From this point forward, we can ask this container to give us the Bars object we want to work with by providing the index value. The syntax for this is:    **BarsArray[index]**    This allows us to get the correct Bars object and use it as input for an [indicator method](https://ninjatrader.com/es/support/helpGuides/nt8/indicators.htm). For example:    **ADX(14)[0] > 30 && ADX(BarsArray[2], 14)[0] > 30**    The above expression in English would translate to:    **If the 14 period ADX of MSFT 1 minute is greater than 30 and the 14 period ADX of AAPL 1 minute is greater than 30**    Before we can apply this concept, we need to ensure that our Bars objects actually contain bars that we can use to run calculations. This can be done by checking the [CurrentBars](https://ninjatrader.com/es/support/helpGuides/nt8/currentbars.htm) array, which returns the number of the current bar in each Bars object. Using this in conjunction with [BarsRequiredToPlot](https://ninjatrader.com/es/support/helpGuides/nt8/barsrequiredtoplot.htm) will ensure each Bars object has sufficient data before we begin processing.     |  | | --- | | **Note**:  By default, the **CurrentBars** starting value will be -1 until all series have processed the first bar. |      | ns | | --- | | protected override void OnBarUpdate() {     // Checks to ensure all Bars objects contain enough bars before beginning.     // If this is a strategy, use BarsRequiredToTrade instead of BarsRequiredToPlot     if (CurrentBars[0] <= BarsRequiredToPlot || CurrentBars[1] <= BarsRequiredToPlot || CurrentBars[2] <= BarsRequiredToPlot)         return; } |     Putting it all together now, the following example checks if the current CCI value for all Bars objects is above 200. You will notice that BarsInProgress is used. This is to check which Bars object is calling the OnBarUpdate() method. More on this later in this section.     | ns | | --- | | protected override void OnBarUpdate() {     // Checks to ensure all Bars objects contain enough bars before beginning     // If this is a strategy, use BarsRequiredToTrade instead of BarsRequiredToPlot     if (CurrentBars[0] <= BarsRequiredToPlot || CurrentBars[1] <= BarsRequiredToPlot || CurrentBars[2] <= BarsRequiredToPlot)         return;       if (BarsInProgress == 0)     {         if (CCI(20)[0] > 200 && CCI(BarsArray[1], 20)[0] > 200           && CCI(BarsArray[2], 20)[0] > 200)         {               // Do something         }     } } | |

[permalink](https://ninjatrader.com/es/support/helpGuides/nt8/index.html?multi-time_frame__instruments.htm#UsingBarsObjectsAsInputToIndicatorMethods)

tog_minus        [True Event Driven OnBarUpdate() Method](javascript:HMToggle('toggle','TrueEventDrivenOnbarupdateMethod','TrueEventDrivenOnbarupdateMethod_ICON'))

|  |  |  |  |
| --- | --- | --- | --- |
| Since a NinjaScript script is truly event driven, the OnBarUpdate() method is called for every bar update event for each Bars object added to the script. This model maximizes flexibility. For example, you could have multiple trading systems combined into one strategy, each dependent on one another. For example, you could have a 1 minute MSFT Bars object and a 1 minute AAPL Bars object, process different trading rules on each Bars object and check to see if MSFT is long when AAPL trading logic is being processed.    The [BarsInProgress](https://ninjatrader.com/es/support/helpGuides/nt8/barsinprogress.htm) property is used to identify which Bars object is calling the OnBarUpdate() method. This allows you to filter out the events that you are looking for.    Continuing our example above, let's take a closer look at what is happening. Remember, we have three Bars objects working in our script, a primary Bars MSFT 1 minute, an MSFT 3 minute, and an AAPL 1 minute.     | ns | | --- | | protected override void OnBarUpdate() {     // Checks to ensure all Bars objects contain enough bars before beginning     // If this is a strategy, use BarsRequiredToTrade instead of BarsRequiredToPlot     if (CurrentBars[0] <= BarsRequiredToPlot || CurrentBars[1] <= BarsRequiredToPlot || CurrentBars[2] <= BarsRequiredToPlot)         return;       // Checks if OnBarUpdate() is called from an update on the primary Bars     if (BarsInProgress == 0)     {         if (Close[0] > Open[0])               // Do something     }       // Checks if OnBarUpdate() is called from an update on MSFT 3 minute Bars     if (BarsInProgress == 1)     {         if (Close[0] > Open[0])               // Do something     }       // Checks if OnBarUpdate() is called from an update on AAPL 1 minute Bars     if (BarsInProgress == 2)     {         if (Close[0] > Open[0])               // Do something     } } |     What is important to understand in the above sample code is that we have "if" branches that check to see what Bars object is calling the OnBarUpdate() method in order to process relevant trading logic. If we only wanted to process the events from the primary Bars we could add the following condition at the top of the OnBarUpdate() method:    **if (BarsInProgress != 0)** **return;**    What is also important to understand is the concept of context. When the OnBarUpdate() method is called, it will be called within the context of the calling Bars object. This means that if the primary Bars triggers the OnBarUpdate() method, all indicator methods and price data will point to that Bars object's data. Notice how the statement "if (Close[0] > Open[0]" exists under each "if" branch in the code sample above. The values returned by Close[0] and Open[0] will be the close and open price values for the calling Bars object. So when the BarsInProgress == 0 (primary Bars) the close value returned is the close price of the MSFT 1 minute bar. When the BarsInProgress == 1 the close value returned is the close price of the MSFT 3 minute Bars object.     |  | | --- | | **Notes:**  •A multi-series script only processes bar update events from the primary Bars (the series the script is applied to) and any additional Bars objects the script adds itself. Additional Bars objects from a multi-series chart or from other multi-series scripts that may be running concurrently will not be processed by this multi-series script.    •If a multi-series script adds an additional Bars object that already exists on the chart, the script will use the preexisting series instead of creating a new one to conserve memory. This includes that series' [session template](https://ninjatrader.com/es/support/helpGuides/nt8/using_the_trading_hours_window.htm) as applied from the chart. If the Bars object does not exist on the chart, the session template of the added Bars object will be the session template of the primary Bars object. If the primary Bars object is using the "<Use instrument settings>" session template, then the additional Bars objects will use the default session templates as defined for their particular instruments in the [Instruments](https://ninjatrader.com/es/support/helpGuides/nt8/instruments.htm) window.    •In a multi-series script, **CurrentBars** starting value will be -1 until all series have processed the first bar. To ensure you have satisfied this requirement on all your Bars objects, it is recommend you start your OnBarUpdate() method with [CurrentBars](https://ninjatrader.com/es/support/helpGuides/nt8/currentbars.htm) checks, as seen in the code sample above.    •A multi-series indicator will hold the same number of data points for plots as the primary series. Setting values to plots should be done in the primary series in OnBarUpdate(). If you are using calculations based off of a larger secondary series, it may plot like a step ladder because there are more data points available than there are actual meaningful data values.    •The default [CloseStrategy()](https://ninjatrader.com/es/support/helpGuides/nt8/closestrategy.htm) handling will only be applied to the primary series of a MultiSeries NinjaScript strategy.    •An indicator / strategy with multiple DataSeries of the same instrument will only process realtime OnBarUpdate() calls when a tick occurs in session of the trading hour templates of all added series.  Any ticks not processed will be queued and processed as a tick comes in for all subsequent DataSeries. | |

[permalink](https://ninjatrader.com/es/support/helpGuides/nt8/index.html?multi-time_frame__instruments.htm#TrueEventDrivenOnbarupdateMethod)

tog_minus        [Accessing the Price Data in a Multi-Bars NinjaScript](javascript:HMToggle('toggle','AccessingThePriceDataInAMultibarsninjascript','AccessingThePriceDataInAMultibarsninjascript_ICON'))

|  |  |  |
| --- | --- | --- |
| As you probably know already, you can access the current bar's closing price with the following statement:    **Close[0];**    You can also access price data such as the close price of other Bars objects at any time. This is accomplished by accessing the [Opens](https://ninjatrader.com/es/support/helpGuides/nt8/opens.htm), [Highs](https://ninjatrader.com/es/support/helpGuides/nt8/highs.htm), [Lows](https://ninjatrader.com/es/support/helpGuides/nt8/lows.htm), [Closes](https://ninjatrader.com/es/support/helpGuides/nt8/closes.htm), [Volumes](https://ninjatrader.com/es/support/helpGuides/nt8/volumeseries.htm), [Medians](https://ninjatrader.com/es/support/helpGuides/nt8/medians.htm), [Typicals](https://ninjatrader.com/es/support/helpGuides/nt8/typicals.htm) and [Times](https://ninjatrader.com/es/support/helpGuides/nt8/timeseries.htm) series by index value. These properties hold collections (containers) that hold their named values for all Bars objects in a script.    Continuing with our example code above, if you wanted to access the high price of the MSFT 3 minute Bars object at index 1 you would write:   **Highs[1][0];**  This is just saying "give me the series of high prices for the Bars object at index 1 'Highs[1]' and return to me the current high value '[0]'". If the BarsInProgress index was equal to 1, the current  context is of the MSFT 3 min Bars object so you could just write:   **High[0];**    The following example demonstrates various ways to access price data.     | ns | | --- | | protected override void OnBarUpdate() {     // Checks to ensure all Bars objects contain enough bars before beginning     // If this is a strategy, use BarsRequiredToTrade instead of BarsRequiredToPlot     if (CurrentBars[0] <= BarsRequiredToPlot || CurrentBars[1] <= BarsRequiredToPlot || CurrentBars[2] <= BarsRequiredToPlot)         return;       // Checks if OnBarUpdate() is called from an update on the primary Bars     if (BarsInProgress == 0)     {         double primaryClose = Close[0];         double msft3minClose = Closes[1][0];         double aapl1minClose = Closes[2][0];           // primaryClose could also be expressed as         // primaryClose = Closes[0][0];     }       // Checks if OnBarUpdate() is called from an update on MSFT 3 minute Bars object     if (BarsInProgress == 1)     {         double primaryClose = Closes[0][0];         double msft3minClose = Close[0];         double aapl1minClose = Closes[2][0];     } } | |

[permalink](https://ninjatrader.com/es/support/helpGuides/nt8/index.html?multi-time_frame__instruments.htm#AccessingThePriceDataInAMultibarsninjascript)

tog_minus        [Entering, Exiting and Retrieving Position Information](javascript:HMToggle('toggle','EnteringExitingAndRetrievingPositionInformation','EnteringExitingAndRetrievingPositionInformation_ICON'))

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| This section is relevant for NinjaScript strategies only. Entry and Exit methods are executed within the BarsInProgress context. Let's demonstrate with an example:     | ns | | --- | | protected override void OnBarUpdate() {     // Checks to ensure all Bars objects contain enough bars before beginning     // If this is an indicator, use BarsRequiredToPlot instead of BarsRequiredToTrade     if (CurrentBars[0] <= BarsRequiredToPlot || CurrentBars[1] <= BarsRequiredToPlot || CurrentBars[2] <= BarsRequiredToPlot)         return;       // Checks if OnBarUpdate() is called from an update on the primary Bars     if (BarsInProgress == 0)     {         // Submits a buy market order for MSFT         EnterLong();     }       // Checks if OnBarUpdate() is called from an update on AAPL 1 minute Bars object     if (BarsInProgress == 2)     {         // Submits a buy market order for AAPL         EnterLong();           // Submits a buy market for MSFT when OnBarUpdate() is called for AAPL         EnterLong(0, 100, "BUY MSFT");     } } |   As you can see above, orders are submitted for MSFT when BarsInProgress is equal to 0 and for AAPL when BarsInProgress is equal to 2. The orders submitted are within the context of the Bars object calling the OnBarUpdate() method and the instrument associated to the calling Bars object. There is one exception, which is the order placed for MSFT within the context of the OnBarUpdate() call for AAPL. Each order method has a variation that allows you to specify the BarsInProgress index value which enables submission of orders for any instrument within the context of another instrument.     |  | | --- | | **Notes**:  1. Should you have multiple Bars objects of the same instrument while using Set() methods in your strategy, you should only submit orders for this instrument to the first Bars context of that instrument. This is to ensure your order logic is processed correctly and any necessary order amendments are done properly.    2. Should you have multiple Bars objects of the same instrument while using options to terminate orders/positions at the end of the session (TIF=Day or [IsExitOnSessionCloseStrategy](https://ninjatrader.com/es/support/helpGuides/nt8/isexitonsessionclosestrategy.htm)=true), you should not submit orders to Bars objects other than the first Bars context for that instrument when on the last bar of the session. This is necessary because some of the end of session handling is applied only to the first Bars context of an instrument, and submitting orders to other Bars objects for that instrument can bypass the end-of-session handling.    3.  For [advanced order methods](https://ninjatrader.com/es/support/helpGuides/nt8/advanced_order_handling.htm), if you **DO NOT** specify a BarsInProgress , the order will be submitted to the current bars in progress updating.  If the current BarsInProgress is a higher time frame, this could delay the time that the order is filled during historical backtesting.  As a result, you should always submit historical orders to the most granular of time frames.    4. When backtesting and submitting orders 'On bar close' and utilizing OnExecutionUpdate or OnOrderUpdate to submit orders, these orders will be processed immediately and filled by the fill engine depending on if the order satisfies its fill condition. This evaluation is done by looking ahead to the next bar of the current series. This is done prior to any secondary higher granularity series having a chance to run its 'OnBarUpdate' logic. If you planned on running order logic in your highest granularity added series then please insure that you submit orders in all cases to the highest granularity series. |     The [Position](https://ninjatrader.com/es/support/helpGuides/nt8/position.htm) property always references the position of the instrument of the current context. If the BarsInProgress is equal to 2 (AAPL 1 minute Bars), Position would refer to the position being held for AAPL. The [Positions](https://ninjatrader.com/es/support/helpGuides/nt8/positions.htm) property holds a collection of Position objects for each instrument in a strategy. Note that there is a critical difference here. Throughout this entire section we have been dealing with Bars objects. Although in our sample we have three Bars objects (MSFT 1 and 3 min and AAPL 1 min) we only have two instruments in the strategy.    **MSFT position is given an index value of 0** **AAPL position is given an index value of 1**    In the example below, when the OnBarUpdate() method is called for the primary Bars we also check if the position held for AAPL is NOT flat and then enter a long position in MSFT. The net result of this strategy is that a long position is entered for AAPL, and then once AAPL is long, we go long MSFT.     | ns | | --- | | protected override void OnBarUpdate() {     // Checks to ensure all Bars objects contain enough bars before beginning     // If this is an indicator, use BarsRequiredToPlot instead of BarsRequiredToTrade     if (CurrentBars[0] <= BarsRequiredToPlot || CurrentBars[1] <= BarsRequiredToPlot || CurrentBars[2] <= BarsRequiredToPlot)         return;       // Checks if OnBarUpdate() is called from an update on the primary Bars     if (BarsInProgress == 0 && Positions[1].MarketPosition != MarketPosition.Flat)     {         // Submits a buy market order for MSFT         EnterLong();     }       // Checks if OnBarUpdate() is called from an update on AAPL 1 minute Bars     if (BarsInProgress == 2)     {         // Submits a buy market order for AAPL         EnterLong();     } } | |

[permalink](https://ninjatrader.com/es/support/helpGuides/nt8/index.html?multi-time_frame__instruments.htm#EnteringExitingAndRetrievingPositionInformation)

|  |  |
| --- | --- |
| **Navigation:**  [NinjaScript](https://ninjatrader.com/es/support/helpGuides/nt8/ninjascript.htm) > [Language Reference](https://ninjatrader.com/es/support/helpGuides/nt8/language_reference_wip.htm) > [Common](https://ninjatrader.com/es/support/helpGuides/nt8/common.htm) > [ISeries<T>](https://ninjatrader.com/es/support/helpGuides/nt8/iseriest.htm) >  **Series<T>** | [Previous page](https://ninjatrader.com/es/support/helpGuides/nt8/iseriest.htm) [Return to chapter overview](https://ninjatrader.com/es/support/helpGuides/nt8/iseriest.htm) [Next page](https://ninjatrader.com/es/support/helpGuides/nt8/reset.htm) |

**Definition**

A Series<T> is a special generic type of data structure that can be constructed with any chosen data type and holds a series of values equal to the same number of elements as bars in a chart. If you have 200 bars loaded in your chart with a moving average plotted, the moving average itself holds a Series<double> object with 200 historical values of data, one for each bar. Series<double> objects can be used as input data for all [indicator methods](https://ninjatrader.com/es/support/helpGuides/nt8/indicators.htm). The Series<T> class implements the ISeries<T> interface.

|  |
| --- |
| **Note**:  By default NinjaTrader limits the number of values stored for Series<T> objects to 256 from the current bar being processed. This drastically improves memory performance by not holding onto old values that are generally not needed. Should you need more values than the last 256 please be sure to create the Series<T> object so that it stores all values instead through the use of the [MaximumBarsLookBack](https://ninjatrader.com/es/support/helpGuides/nt8/maximumbarslookback.htm) property. |

**Parameters**

|  |  |
| --- | --- |
| ninjaScriptBase | The NinjaScript object used to create the Series |
| bars | The [Bars](https://ninjatrader.com/es/support/helpGuides/nt8/bars.htm) object used to create the Series |
| maximumBarsLookBack | A [MaximumBarsLookBack](https://ninjatrader.com/es/support/helpGuides/nt8/maximumbarslookback.htm) value used for memory performance |

**Methods and Properties**

|  |  |
| --- | --- |
| [GetValueAt()](https://ninjatrader.com/es/support/helpGuides/nt8/getvalueat.htm) | Returns the underlying input value at a specified bar index value. |
| [IsValidDataPoint()](https://ninjatrader.com/es/support/helpGuides/nt8/isvaliddatapoint.htm) | Determines if the specified input is set at a barsAgo value relative to the current bar. |
| [Reset()](https://ninjatrader.com/es/support/helpGuides/nt8/reset.htm) | Resets the internal marker which is used for [IsValidDataPoint()](https://ninjatrader.com/es/support/helpGuides/nt8/isvaliddatapoint.htm) back to false. |
| [Count](https://ninjatrader.com/es/support/helpGuides/nt8/count.htm) | The total number of bars or data points. |

**Creating Series<T> Objects**

When creating custom indicators, Series<double> objects are automatically created for you by calling the [AddPlot()](https://ninjatrader.com/es/support/helpGuides/nt8/addplot.htm) method and can be subsequently referenced by the [Value](https://ninjatrader.com/es/support/helpGuides/nt8/value.htm) and/or [Values](https://ninjatrader.com/es/support/helpGuides/nt8/values.htm) property. However, you may have a requirement to create a Series<T> object to store values that are part of an overall indicator value calculation. This can be done within a custom indicator or strategy.

|  |
| --- |
| **Note**:  Custom Series<T> objects will hold the number of values specified by the [MaximumBarsLookBack](https://ninjatrader.com/es/support/helpGuides/nt8/maximumbarslookback.htm) property when the custom series object is instantiated. |

To create a Series<T> object:

1.Determine the data type of the Series<T> object you wish to create. This could be double, bool, int, string or any other object type you want.

2.Define a variable of type Series<T> that will hold a Series<T> object. This example will create "myDoubleSeries" as a Series<double>.

3.In the [OnStateChange()](https://ninjatrader.com/es/support/helpGuides/nt8/onstatechange.htm) method, in the State.DataLoaded create a new Series<T> object and assign it to the "myDoubleSeries" variable

| ns |
| --- |
| private Series<double> myDoubleSeries; // Define a Series<T> variable. In this instance we want it                                       // as a double so we created a Series<double> variable.   // Create a Series object and assign it to the variable protected override void OnStateChange() {     if (State == State.DataLoaded)     {         // "this" refers to the NinjaScript object itself. This syncs the Series object to historical data bars         // MaximumBarsLookBack determines how many values the Series<double> will have access to         myDoubleSeries = new Series<double>(this, MaximumBarsLookBack.Infinite);     } } |

|  |
| --- |
| **Tip***:*Series<T> objects can be used on supplementary series in a multi-time frame and instrument strategy. Please see our [support forum](http://www.ninjatrader.com/support/forum/showthread.php?t=3572) NinjaScript reference samples section for further information. |

**Setting Values**

You can set the value for the current bar being evaluated by choosing a "barsAgo" value of "0" or, for historical bars, by choosing a "barsAgo" value that represents the number of bars ago that you want the value to be stored at.

| ns **Setting Series<T> values** |
| --- |
| protected override void OnBarUpdate() {     myDoubleSeries[0] = Close[0]; } |

|  |
| --- |
| **Note**:  The "barsAgo" value is only guaranteed to be in sync with the recent current bar during core data event methods, such as OnBarUpdate(), OnMarketUpdate(), and during strategy related order events such as OnOrderUpdate(), OnExecutionUpdate(), OnPositionUpdate().  For scenarios where you may need to set a value outside of a core data/order event, such as OnRender() or a custom event, you must first synchronize the "barsAgo" pointer via the [TriggerCustomEvent()](https://ninjatrader.com/es/support/helpGuides/nt8/triggercustomevent.htm) method. |

**Checking for Valid Values**  
It is possible that you may use a Series<T> object but decide not to set a value for a specific bar. However, you should *not* try to access a Series<T>value that has not been set. Internally, a dummy value does exists, but you want to check to see if it was a valid value that you set before trying to access it for use in your calculations.  Please see [IsValidDataPoint()](https://ninjatrader.com/es/support/helpGuides/nt8/isvaliddatapoint.htm) more information.

|  |
| --- |
| **Warning**:  Calling IsValidDataPoint() will only work a [MaximumBarsLookBackInfinite](https://ninjatrader.com/es/support/helpGuides/nt8/maximumbarslookback.htm) series.  Attempting to check IsValidDataPoint() MaximumBarsLookBack256 series throw an error.  Please check the Log tab of the Control Center |

**Getting Values**  
You can access Series<T> object values using the syntax Series<T>[int *barsAgo*] where barsAgo represents the data value *n* (number of bars ago).

| ns **Accessing Series object values** |
| --- |
| protected override void OnBarUpdate() {   // Prints the current and last bar value   Print("The values are " + myDoubleSeries[0] + " " + myDoubleSeries[1]); } |

Alternatively, you can access a value at an absolute bar index using the [GetValueAt()](https://ninjatrader.com/es/support/helpGuides/nt8/getvalueat.htm) method.

|  |
| --- |
| **Note**:  In most cases, you will access the historical price series using a core data event handler such as OnBarUpdate().  For more advance developers, you may find situations where you wish to access historical price series outside of the core data event methods, such as OnRender(), or your own custom event.  In these advanced scenarios, you may run into situations where the "barsAgo" pointer is not in sync with the current bar, and may result in errors when trying to obtain this information.  In those cases, please use the Bars.Get...() methods with the absolute bar index, e.g., [GetValueAt()](https://ninjatrader.com/es/support/helpGuides/nt8/getvalueat.htm). |

**Methods that Accept ISeries<T> as Arguments**  
All [indicator methods](https://ninjatrader.com/es/support/helpGuides/nt8/indicators.htm) accept ISeries<double> objects as arguments. Carrying from the prior examples, let's print out the 10 period simple moving average of range.

| ns **Using a custom Series object as indicator input** | |
| --- | --- |
| protected override void OnBarUpdate() {   // Calculate the range of the current bar and set the value     myDoubleSeries[0] = (High[0] - Low[0]);       // Print the current 10 period SMA of range     Print("Value is " + SMA(myDoubleSeries, 10)[0]);         } | |
| **Navigation:**  [NinjaScript](https://ninjatrader.com/es/support/helpGuides/nt8/ninjascript.htm) > [Educational Resources](https://ninjatrader.com/es/support/helpGuides/nt8/educational_resources.htm) >  **NinjaScript Lifecycle** | | [Previous page](https://ninjatrader.com/es/support/helpGuides/nt8/multi-time_frame__instruments.htm) [Return to chapter overview](https://ninjatrader.com/es/support/helpGuides/nt8/educational_resources.htm) [Next page](https://ninjatrader.com/es/support/helpGuides/nt8/using_3rd_party_indicators.htm) |

NinjaTrader uses a [State](https://ninjatrader.com/es/support/helpGuides/nt8/state.htm) change system to represent various life cycles of your NinjaScript object.  For more basic indicators and strategies, simply understanding each **State** described on the [OnStateChange()](https://ninjatrader.com/es/support/helpGuides/nt8/onstatechange.htm) page is sufficient.  However, for more advanced development projects, it is critical to understand how NinjaTrader calls these states for various instances throughout the lifetime of the entire application.

**When NinjaTrader instantiates a NinjaScript object**

There are two categories of instances instantiated by NinjaTrader:

•"UI" instances representing its default properties on various user interfaces

•The "configured" instance executing your custom instructions

In both categories, [OnStateChange()](https://ninjatrader.com/es/support/helpGuides/nt8/onstatechange.htm) is called at least twice:  once to **State.SetDefaults** acquiring various default property values, and then again to **State.Terminated** handling internal references cleanup.

|  |
| --- |
| **Note**:  It is important to understand that previous major versions of NinjaTrader were not so diligent in running termination logic for UI instances and the current major NinjaTrader 8 version has been changed to help properly address related issues. |

To elaborate on that process, imagine the sequence of user events required to start an indicator on a chart:

1.User right clicks on a Chart and select "**Indicator**"

2.User adds an Indicator from the **Available** list

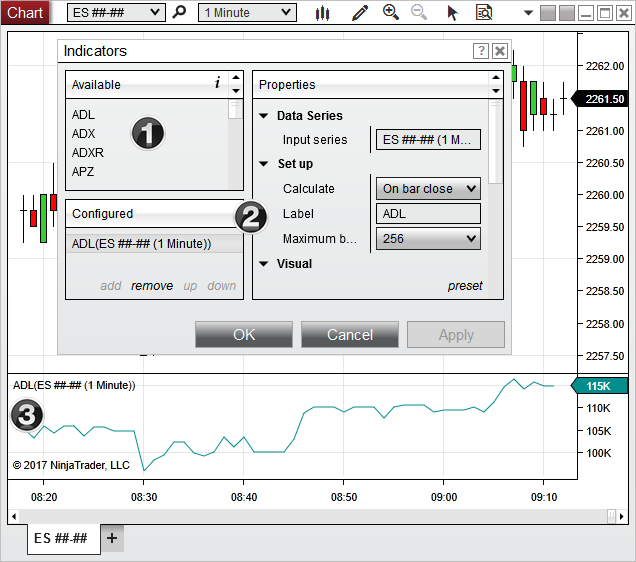
3.User configures desired **Properties** and presses "**Apply**" or "**OK**"

During this sequence, there are actually 3 instances of the same indicator created by NinjaTrader:

1.The instance displaying the **Name** property to the list of "**Available**" indicators (**Note**: this process involves creating an instance of *all* indicators in order to build the complete list)

2.The instance displaying the individual **Name** and its default **Properties**

3.The instance configured and executing on the chart



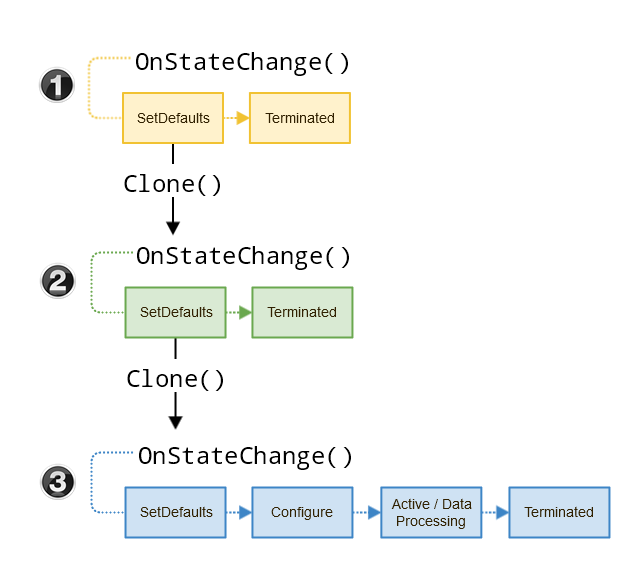
To visualize how each instance goes through its **States**, please consider the logic and flow chart below:

1.In order to display the indicator name in the list of **"Available"** indicators, the NinjaTrader core must find the **Name** of each installed indicator defined in their **SetDefaults**.  This occurs simultaneously for *every indicator installed on the system* in order to build the full list of available indicators.

2.The selected indicator is then [cloned](https://ninjatrader.com/es/support/helpGuides/nt8/clone.htm) and **SetDefaults** is called again in order to display the default properties to the "**Properties**" grid.  This only occurs for the individual indicator.

3.After the user has set their desired property settings and press **OK** or **Apply**, the indicator is once again cloned and runs through its full state management.  This only occurs for the indicator configured to execute on the chart.

|  |
| --- |
| **Warning**:  Since NinjaTrader is multi-threaded, it is possible the **OnStateChange()** logic will be operating on a different thread than your indicator instances.  Due to this fact, if logic in your **OnStateChange()** method is thread sensitivity (e.g., dependent on UI threads vs Instrument threads) please make sure to read the section on [multi-threading considerations](https://ninjatrader.com/es/support/helpGuides/nt8/multi-threading.htm) and check for thread access in your **OnStateChange()** logic |



It is the 3rd "configured" instance you are concerned with developing, but you should also be aware of the "UI" instances which are triggered at various stages of NinjaTrader.

|  |
| --- |
| **Notes**:  1.The example above is written for an indicator, but the same concept of state management applies to every NinjaScript object type  2.The UI instances do not reach **State.Terminated** until the user closes out of the UI feature displaying the object  3.Since [AddOns](https://ninjatrader.com/es/support/helpGuides/nt8/addon_development_overview.htm) run in the background and are not dependent on UI elements, they will run through their **SetDefaults**/**Terminated** states after each NinjaScript compile and startup/shutdown of NinjaTrader.  4.The configured instance will also be cloned back to UI instances during various user actions (e.g, re-opening an indicator dialog to reconfigure settings, or user copying & pasting the indicator to a new panel or chart).  Therefore you should not assume that objects (such as ChartControl) will not be accessible in the UI instances.  5.In some extreme scenarios, you may need to execute custom logic before or after an object is cloned.  Overriding the default behavior can be done via the virtual [Clone()](https://ninjatrader.com/es/support/helpGuides/nt8/clone.htm) method |

**What does this mean for me?**

Since **OnStateChange()** can be called at various times throughout NinjaTrader, you should be diligent in handling each state and managing resources only when it is appropriate that your NinjaScript object was actually configured:

•**State.SetDefaults** should be kept as lean as possible to prevent logic from processing superfluously and causing problems unrelated to the configured instance.  Only properties which need to be displayed on the UI should be set in this state.

•Resources should only be set up once an object has reached **State.Configure** or **State.DataLoaded** (see [best practices](https://ninjatrader.com/es/support/helpGuides/nt8/ninjascript_best_practices.htm) for more information)

•**State.Terminated** logic should be specific in when it resets a value or destroys a resource.  Since the running instance can be cloned back to a UI instance, checking that a mutable property exists before accessing sometimes is not enough.  You may need to consider adding a flag to help decide when a resource needs to be reset or destroyed.

**Example**

Let’s say your object was an indicator looking to add a custom toolbar element to the chart, and when the indicator is removed from the chart, you would want to make sure your toolbar elements are also properly removed.  In the [OnStateChange()](https://ninjatrader.com/es/support/helpGuides/nt8/onstatechange.htm) handler you change could add the custom toolbar once the **State** has reached **State.Historical**, and remove the toolbar once the State has reached **State.Terminated**.

To ensure that the remove logic only runs in instances that were actually configured, you can see we in the example below we also track that the toolbar needs a reset in **State.Terminated** state via a custom bool variable.  In other words, the proper reset request comes from our configured instance and would be ignored if the **State.Terminated** is called from outside our object (i.e., a UI instance). This will prepare our object to properly handle both situations in which **State.Terminated** could be called in the NinjaTrader state management system.

| ns |
| --- |
| // custom flag to help time termination logic private bool toolBarNeedsReset = false;   protected override void OnStateChange() {   if (State == State.SetDefaults)   {     Name = "State lifetime indicator";   }   else if (State == State.Historical)   {     // before indicator starts historical processing     // add a custom tool bar using a custom method     AddToolBarButton(); // this is a pseudo-method for example purposes     toolBarNeedsReset = true; // use a flag to track this logic was executed   }     else if (State == State.Terminated)   {     // here we intend to remove the custom tool bar when the indicator shuts down     if (toolBarNeedsReset) // flag is only true after actually added         RemoveToolBarButton();   } } |

**Cloning NinjaScript**

Clone is the operation of iterating over all public browsable properties on a NinjaScript object and duplicating the values over to a freshly generated instance. For the majority of NinjaScript with standard properties the clone process is transparent to you and you do not need to be concerned the the clone process. For those of you that want more control or will be utilizing complex properties then knowledge about clone is essential. Cloning is performed in 2 primary use cases:

1.Configuring an instance in an object dialog and then cloning the configured data to an actual NinjaScript instance applied for example to a Chart. (Configuration then Run)

2.When triggering 'Reload NinjaScript' or "Reload All Historical Data'

NinjaScript objects have a base clone method implemented which will iterating over all browsable properties and copy by value to the next instance. The rules follow the 'clone' rules described in the clone documentation located [here](https://ninjatrader.com/es/support/helpGuides/nt8/clone.htm) and described above. The default behavior will work in almost all cases except for when you have some complex custom property which needs specific clone behavior. In which case we allow the ability to override Clone() and specify your own behavior.

|  |
| --- |
| **Note**: If you plan to utilize complex class properties on NinjaScript, you can specify your own clone method. However when NinjaScript is compiled in NinjaTrader a new DLL holding the compiled IL code is 'hot-loaded' into NinjaTrader. As a user or developer would try to reload NinjaScript or configure an existing NinjaScript object, any complex class will not resolve since the class will be residing in two different assemblies. This problem cannot be solved with custom clone method and workarounds for this are setting Browsable(false) attribute on that property so it is not cloned or putting the property it its own dedicated assembly. |

**Saving NinjaScript Properties to the Workspace via XML Serialization**

XML Serialization comes into play when you have a set of properties and want those properties to persist the user saved workspace (or any templates that are user created).

By default basic types such as int, string, bool will all serialize without issue, if you have a complex property you want its setting maintained on restore then you need to create a string serialized representation of that class. The technique is shown in this example post [here](https://ninjatrader.com/es/support/helpGuides/nt8/user_definable_color_inputs.htm) where we show how to serialize a color brush.

|  |  |
| --- | --- |
| **Navigation:**  [NinjaScript](https://ninjatrader.com/es/support/helpGuides/nt8/ninjascript.htm) > [Language Reference](https://ninjatrader.com/es/support/helpGuides/nt8/language_reference_wip.htm) > [Common](https://ninjatrader.com/es/support/helpGuides/nt8/common.htm) > [System Indicator Methods](https://ninjatrader.com/es/support/helpGuides/nt8/indicators.htm) >  **ZigZag** | [Previous page](https://ninjatrader.com/es/support/helpGuides/nt8/woodies_pivots.htm) [Return to chapter overview](https://ninjatrader.com/es/support/helpGuides/nt8/indicators.htm) [Next page](https://ninjatrader.com/es/support/helpGuides/nt8/tradinghours.htm) |

**Description**

The ZigZag indicator highlights trends based on user defined threshold values and helps filtering the noise in price charts, it's not a classical indicator but more a reactive filter showing extreme price points. In processing it's calculations it can update it's current direction and price extreme point based on newly incoming data, the current developing leg should be thought of temporary until a new leg in opposite direction has been set.

You can access methods within this indicator to determine the number of bars ago a zigzag high or low point occurred or the current zigzag value, it is only meaningful to work with in Calculate.OnBarClose mode for the [Calculate](https://ninjatrader.com/es/support/helpGuides/nt8/calculate.htm) property.

**Syntax - Bars Ago**

High Bar  
ZigZag(DeviationType *deviationType*, double *deviationValue*, bool *useHighLow*).HighBar(int *barsAgo*, int *instance*, int *lookBackPeriod*)  
ZigZag(ISeries<double> *input*, DeviationType *deviationType*, double *deviationValue*, bool *useHighLow*).HighBar(int *barsAgo*, int *instance*, int *lookBackPeriod*)  
   
Low Bar  
ZigZag(DeviationType *deviationType*, double *deviationValue*, bool *useHighLow*).LowBar(int *barsAgo*, int *instance*, int *lookBackPeriod*)  
ZigZag(ISeries<double> *input*, DeviationType *deviationType*, double *deviationValue*, bool *useHighLow*).LowBar(int *barsAgo*, int *instance*, int *lookBackPeriod*)

**Return Value**

An int value representing the number of bars ago. Returns a value of -1 if a swing point is not found within the look back period.

**Syntax - Value**

|  |
| --- |
| High Value ZigZag(DeviationType *deviationType*, double *deviationValue*, bool *useHighLow*).ZigZagHigh[int *barsAgo*] ZigZag(ISeries<double> *input*, DeviationType *deviationType*, double *deviationValue*, bool *useHighLow*).ZigZagHigh[int *barsAgo*]   Low Value ZigZag(DeviationType *deviationType*, double *deviationValue*, bool *useHighLow*).ZigZagLow[int *barsAgo*] ZigZag(ISeries<double> *input*, DeviationType *deviationType*, double *deviationValue*, bool *useHighLow*).ZigZagLow[int *barsAgo*] |

**Return Value**

double; Accessing this method via an index value [int *barsAgo*] returns the indicator value of the referenced bar.

**\* A return value of 0 (zero) indicates that a zigzag high or low has not yet formed.**

**Parameters**

|  |  |
| --- | --- |
| barsAgo | The number of bars ago that serves as the starting bar and works backwards |
| deviationType | Possible values are: DeviationType.Points DeviationType.Percent |
| deviationValue | The deviation value |
| input | Indicator source data ([?](https://ninjatrader.com/es/support/helpGuides/nt8/valid_input_data_for_indicator.htm)) |
| instance | The occurrence to check for (1 is the most recent, 2 is the 2nd most recent etc...) |
| lookBackPeriod | Number of bars to look back to check for the test condition. Test is evaluated on the current bar and the bars in the look back period. |
| useHighLow | When true, both High and Low price series are used. When false, the default input is used for both highs and lows. |

**Example**

| ns |
| --- |
| // Prints the high price of the most recent zig zag high Print("The high of the zigzag bar is " + High[Math.Max(0, ZigZag(DeviationType.Points, 0.5, false).HighBar(0, 1, 100))]); |

**Source Code**

You can view this indicator method source code by selecting the menu **New > NinjaScript Editor > Indicators** within the NinjaTrader Control Center window.

|  |  |
| --- | --- |
| **Navigation:**  »No topics above this level«  **Entering Calculation Logic** | [Return to chapter overview](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_directwrite_textlayout.htm) |

The [OnBarUpdate()](https://ninjatrader.com/es/support/helpGuides/nt8/onbarupdate.htm) method is called for each incoming tick, or on the close of a bar (if enabled) when performing real-time calculations, and is called on each bar of a [Bars](https://ninjatrader.com/es/support/helpGuides/nt8/bars.htm) object when re-calculating the indicator (For example, an indicator would be re-calculated when adding it to an existing chart that has existing price data displayed). This is the main method called for indicator calculation, and we will calculate the CCI value and set the conditions used to draw the CCI plot within this method.

The [OnStateChange()](https://ninjatrader.com/es/support/helpGuides/nt8/onstatechange.htm) method is called once before any bar data is loaded, and is used to configure the indicator (among other things).

**Initializing the Indicator**  
The code below is automatically generated by the wizard and added to the OnStateChange() method, within State.SetDefaults. It configures the indicator for one plot and five lines, and sets the parameters entered in the wizard:

| ns |
| --- |
| AddLine(Brushes.DimGray, 200, "Level 2"); AddLine(Brushes.DimGray, 100, "Level 1"); AddLine(Brushes.DimGray, 0, "Zero Line"); AddLine(Brushes.DimGray, -100, "Level -1"); AddLine(Brushes.DimGray, -200, "Level -2"); |

To change the visual properties of the Zero Line, replace the third line in the code above with the line below. This will change the color to black and the line style to "dash:"

| ns |
| --- |
| AddLine(new Stroke(Brushes.Black, DashStyleHelper.Dash,2), 0, "Zero Line"); |

The code above uses an alternative method overload (an alternative set of arguments passed in to the AddLine() method), in order to pass in a [Stroke](https://ninjatrader.com/es/support/helpGuides/nt8/stroke_class.htm) object rather than a [Brush](http://sharpdx.org/documentation/api/t-sharpdx-direct2d1-brush). With a Stroke, not only can we still specify a Brush, but we have additional options to change the dash style (via DashStyleHelper) and the line width. After this change, your configured lines and plots should look like this:

| ns |
| --- |
| AddLine(Brushes.DimGray, 200, "Level 2"); AddLine(Brushes.DimGray, 100, "Level 1"); AddLine(new Stroke(Brushes.Black, DashStyleHelper.Dash,2), 0, "Zero Line"); AddLine(Brushes.DimGray, -100, "Level -1"); AddLine(Brushes.DimGray, -200, "Level -2"); |

**Adding Core Indicator Logic**

Since this tutorial is meant to cover custom drawing and manually changing properties within an indicator, we will not go too in-depth into the core calculation logic for this custom CCI. Instead, we will copy and paste the core calculation logic from the @CCI indicator already built-in to NinjaTrader.

The @CCI indicator uses an SMA object in its calculations. To add this, copy the line below from @CCI into your custom CCI, directly below the class declaration:

| ns |
| --- |
| private SMA sma; |

Next, copy the following initialization for the SMA object into the OnStateChange() method, within State.Configure:

| ns |
| --- |
| sma = SMA(Typical, Period); |

Next, copy the core calculation logic from @CCI into the OnBarUpdate() method of your custom indicator:

| ns |
| --- |
| if (CurrentBar == 0)   Value[0] = 0; else {   double mean = 0;   double sma0 = sma[0];     for (int idx = Math.Min(CurrentBar, Period - 1); idx >= 0; idx--)       mean += Math.Abs(Typical[idx] - sma0);     Value[0] = (Typical[0] - sma0) / (mean.ApproxCompare(0) == 0 ? 1 : (0.015 \* (mean / Math.Min(Period, CurrentBar + 1)))); } |

The code for your MyCCI class should now look as follows (in addition to the using statements and class declaration) :

| ns |
| --- |
| public class MyCCI : Indicator {     private SMA sma;     protected override void OnStateChange()   {       if (State == State.SetDefaults)       {           Description                 = @"NinjaScript Custom Drawing Indicator Tutorial";           Name                       = "MyCCI";           Calculate                   = Calculate.OnBarClose;           IsOverlay                   = false;           DisplayInDataBox           = true;           DrawOnPricePanel           = true;           DrawHorizontalGridLines     = true;           DrawVerticalGridLines       = true;           PaintPriceMarkers           = true;           ScaleJustification         = NinjaTrader.Gui.Chart.ScaleJustification.Right;           //Disable this property if your indicator requires custom values that cumulate with each new market data event.           //See Help Guide for additional information.           IsSuspendedWhileInactive   = true;           Period                 = 14;           AddPlot(Brushes.Orange, "CCI");           AddLine(Brushes.DimGray, 200, "Level 2");           AddLine(Brushes.DimGray, 100, "Level 1");           AddLine(new Stroke(Brushes.Black, DashStyleHelper.Dash,2), 0, "Zero Line");           AddLine(Brushes.DimGray, -100, "Level -1");           AddLine(Brushes.DimGray, -200, "Level -2");         }       else if (State == State.Configure)       {           sma = SMA(Typical, Period);       }   }     protected override void OnBarUpdate()   {       if (CurrentBar == 0)           Value[0] = 0;       else       {           double mean = 0;           double sma0 = sma[0];             for (int idx = Math.Min(CurrentBar, Period - 1); idx >= 0; idx--)               mean += Math.Abs(Typical[idx] - sma0);             Value[0] = (Typical[0] - sma0) / (mean.ApproxCompare(0) == 0 ? 1 : (0.015 \* (mean / Math.Min(Period, CurrentBar + 1))));       }   } |

**Custom Drawing**  
Add the following code into the OnBarUpdate() method, directly beneath the core calculation logic:

| ns |
| --- |
| // if the plot value is greater than 100, paint the plot green at that bar index if (Value[0] > 100)   PlotBrushes[0][0] = Brushes.Green;   // if the plot value is less than -100, paint the plot red at that bar index if (Value[0] < -100)   PlotBrushes[0][0] = Brushes.Red;   // if the plot value is between 100 and -100, paint the plot orange at that bar index if (Value[0] >= -100 && Value[0] <= 100)   PlotBrushes[0][0] = Brushes.Orange; |

This will conditionally change the color of the CCI plot (referenced by Values[0]) based on its value. By using PlotBrushes[0][0], we are specifying that we wish to change the color of the first plot in the collection at a specific bar index (the current bar index each time the condition is triggered), and we wish for the plot the remain that color at that index, even if the plot value changes in the future. If instead we wished to change the entire plot color, we could use Plots[0].Brush.

PlotBrushes holds a collection of brushes used for the various plots in the indicator. In addition to this, there are several other collections that serve similar purposes, which can be used in the same way. Some examples of these collections are below:

|  |  |
| --- | --- |
| [BackBrushes](https://ninjatrader.com/es/support/helpGuides/nt8/backbrushes.htm) | A collection of Brushes used for chart background color at specific bar indexes |
| [BarBrushes](https://ninjatrader.com/es/support/helpGuides/nt8/barbrushes.htm) | A collection of Brushes used to paint bars at specific indexes |
| [CandleOutlineBrushes](https://ninjatrader.com/es/support/helpGuides/nt8/candleoutlinebrushes.htm) | A collection of Brushes used to paint candle outlines at specific indexes |

Now that everything is in place, your class code should look as below. You are now ready to [compile the indicator](https://ninjatrader.com/es/support/helpGuides/nt8/compiling6.htm) and configure it on a chart.

| ns | |
| --- | --- |
| public class MyCCI : Indicator {     private SMA sma;     protected override void OnStateChange()   {       if (State == State.SetDefaults)       {           Description                 = @"NinjaScript Custom Drawing Indicator Tutorial";           Name                       = "MyCCI";           Calculate                   = Calculate.OnBarClose;           IsOverlay                   = false;           DisplayInDataBox           = true;           DrawOnPricePanel           = true;           DrawHorizontalGridLines     = true;           DrawVerticalGridLines       = true;           PaintPriceMarkers           = true;           ScaleJustification         = NinjaTrader.Gui.Chart.ScaleJustification.Right;           //Disable this property if your indicator requires custom values that cumulate with each new market data event.           //See Help Guide for additional information.           IsSuspendedWhileInactive   = true;           Period                 = 14;           AddPlot(Brushes.Orange, "CCI");           AddLine(Brushes.DimGray, 200, "Level 2");           AddLine(Brushes.DimGray, 100, "Level 1");           AddLine(new Stroke(Brushes.Black, DashStyleHelper.Dash,2), 0, "Zero Line");           AddLine(Brushes.DimGray, -100, "Level -1");           AddLine(Brushes.DimGray, -200, "Level -2");         }       else if (State == State.Configure)       {           sma = SMA(Typical, Period);       }   }     protected override void OnBarUpdate()   {       if (CurrentBar == 0)           Value[0] = 0;       else       {           double mean = 0;           double sma0 = sma[0];             for (int idx = Math.Min(CurrentBar, Period - 1); idx >= 0; idx--)               mean += Math.Abs(Typical[idx] - sma0);             Value[0] = (Typical[0] - sma0) / (mean.ApproxCompare(0) == 0 ? 1 : (0.015 \* (mean / Math.Min(Period, CurrentBar + 1))));       }         if (Value[0] > 100)           PlotBrushes[0][0] = Brushes.Green;         if (Value[0] < -100)           PlotBrushes[0][0] = Brushes.Red;         if (Value[0] >= -100 && Value[0] <= 100)           PlotBrushes[0][0] = Brushes.Orange;   } | |
| **Navigation:**  [NinjaScript](https://ninjatrader.com/es/support/helpGuides/nt8/ninjascript.htm) > [SharpDX SDK Reference](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_sdk_reference.htm) > [SharpDX.Direct2D1](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1.htm) >  **SharpDX.Direct2D1.RenderTarget** | | [Previous page](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_radialgradientbrushproperties.htm) [Return to chapter overview](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1.htm) [Next page](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_rendertarget_antialiasmode.htm) |
| **Disclaimer**: The [SharpDX SDK Reference](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_sdk_reference.htm) section was compiled from the official [SharpDX Documentation](http://sharpdx.org/) and was **NOT** authored by NinjaTrader.  The contents of this section are provided as-is and only cover a fraction of what is available from the **SharpDX SDK**.  This page was intended only as a reference guide to help you get started with some of the 2D Graphics concepts used in the **NinjaTrader.Custom** assembly.  Please refer to the official **SharpDX Documentation** for additional members not covered in this reference.  For more seasoned graphic developers, the original**MSDN** [Direct2D1](https://msdn.microsoft.com/en-us/library/windows/desktop/dd370990.aspx) and [DirectWrite](https://msdn.microsoft.com/en-us/library/windows/desktop/dd368038.aspx) unmanaged API documentation can also be helpful for understanding the DirectX/Direct2D run-time environment. *For NinjaScript development purposes, we document only****essential****members in the structure of this page.* | | |

**Definition**

Represents an object that can receive drawing commands.

(See also [unmanaged API documentation](http://msdn.microsoft.com/en-us/library/dd371766.aspx))

**Syntax**

class RenderTarget

|  |
| --- |
| **Tips**:  1.For NinjaScript Development purposes, [DrawingTools](https://ninjatrader.com/es/support/helpGuides/nt8/drawingtool.htm), [ChartStyles](https://ninjatrader.com/es/support/helpGuides/nt8/chartstyletype.htm), [Indicators](https://ninjatrader.com/es/support/helpGuides/nt8/indicators.htm), and [Strategies](https://ninjatrader.com/es/support/helpGuides/nt8/strategy.htm) implement the Chart's [RenderTarget](https://ninjatrader.com/es/support/helpGuides/nt8/rendertarget.htm) ready to be used in the **OnRender()** method  2.General information on **Direct2D** **Render Targets** can be found on the [MSDN Direct2D Render Targets Overview](https://msdn.microsoft.com/en-us/library/dd756757(v=vs.85).aspx) |

**Methods and Properties**

|  |  |
| --- | --- |
| [AntialiasMode](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_rendertarget_antialiasmode.htm) | Retrieves or sets the current antialiasing mode for nontext drawing operations. |
| [DrawEllipse()](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_rendertarget_drawellipse.htm) | Draws the outline of the specified ellipse using the specified stroke style. |
| [DrawGeometry()](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_rendertarget_drawgeometry.htm) | Draws the outline of the specified geometry. |
| [DrawLine()](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_rendertarget_drawline.htm) | Draws a line between the specified points. |
| [DrawRectangle()](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_rendertarget_drawrectangle.htm) | Draws the outline of a rectangle that has the specified dimensions. |
| [DrawText()](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_rendertarget_drawtext.htm) | Draws the specified text using the format information provided by an [SharpDX.DirectWrite.TextFormat](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_directwrite_textformat.htm) object. |
| [DrawTextLayout()](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_rendertarget_drawtextlayout.htm) | Draws the formatted text described by the specified [SharpDX.DirectWrite.TextLayout](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_directwrite_textlayout.htm) object. |
| [FillEllipse()](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_rendertarget_fillellipse.htm) | Paints the interior of the specified ellipse. |
| [FillGeometry()](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_rendertarget_fillgeometry.htm) | Paints the interior of the specified geometry. |
| [FillRectangle()](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_rendertarget_fillrectangle.htm) | Paints the interior of the specified rectangle. |
| [IsDisposed](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_disposebase_isdisposed.htm) | Gets a value indicating whether this instance is disposed.  (Inherited from [SharpDX.DisposeBase](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_disposebase.htm).) |
| [Transform](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_direct2d1_rendertarget_transform.htm) | Gets or sets the current transform of the render target. |