|  |  |
| --- | --- |
| **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) >  **IsInstantiatedOnEachOptimizationIteration** | [Previous page](https://ninjatrader.com/es/support/helpGuides/nt8/isfilllimitontouch.htm) [Return to chapter overview](https://ninjatrader.com/es/support/helpGuides/nt8/strategy.htm) [Next page](https://ninjatrader.com/es/support/helpGuides/nt8/istradinghoursbreaklinevisible.htm) |

**Definition**

Determines if the strategy should be re-instantiated (re-created) after each optimization run when using the [Strategy Analyzer Optimizer](https://ninjatrader.com/es/support/helpGuides/nt8/optimize_a_strategy.htm).

The **default behavior** is to re-instantiate the strategy for each optimization backtest run. However, the process of re-instantiating a strategy requires more time and computer resources to return results, which could impact the amount of time it takes to run an optimization.  When **false**, the strategy is re-used to save time and computer resources.  Under this design, internal properties are reset to default values after each iteration, but it is possible that user-defined properties and other custom resources may carry their state over from the previous iteration into a new backtest run.  To take advantage of performance optimizations, developers may need to reset class level variables in the strategy otherwise unexpected results can occur.

|  |
| --- |
| **Note**:  If you choose to take advantage of the performance benefits during strategy optimization by setting the **IsInstantiatedOnEachOptimizationIteration** property to **false**, any objects you create in your code **MUST** be reset duringthe appropriate**State**within the [OnStateChange()](https://ninjatrader.com/es/support/helpGuides/nt8/onstatechange.htm)method.  Please see the example below on "*Manually resetting class level variables to take advantage of Strategy Analyzer optimizer performance benefits*". |

**Property Value**

This property returns **true** if the strategy is not recycled; otherwise, **false**. Default set to **true**.

|  |
| --- |
| **Warning**:  This property should **ONLY** bet set from the [OnStateChange()](https://ninjatrader.com/es/support/helpGuides/nt8/onstatechange.htm) method during **State.SetDefaults** or **State.Configure** |

**Syntax**

IsInstantiatedOnEachOptimizationIteration

|  |
| --- |
| **Tip**:  The default NinjaTrader indicators and strategies have been optimized to take advantage of performance optimizations as their resources are setup >= **State.Configure**.  Please see the default system indicators and strategies for an idea of how you may improve your strategy and indicator performance, or you may also reference the example code below. |

**Examples**

| ns **Using IsInstantiatedOnEachOptimizationIteration to reset class level variables** |
| --- |
| // A custom trades dictionary is created when strategy is instantiated // since we later set "IsInstantiatedOnEachOptimizationIteration" to true, // we are guaranteed to start with a new object on each optimization run private Dictionary<DateTime, string> myTrades = new Dictionary<DateTime, string>();   protected override void OnStateChange() {   if (State == State.SetDefaults)   {     Name       = "My Optimization Test 1";     Description = "Demonstrates using IsInstantiatedOnEachOptimizationIteration to reset a class level variable";     Fast       = 10;     Slow       = 25;       // setting to true so our custom trades dictionary is reset on each optimization run (comes with a performance penalty)     // This is the default behavior.     IsInstantiatedOnEachOptimizationIteration = true;   }     else if (State == State.Terminated)   {     // Print the number of trades at the end of the optimization     if (myTrades != null)     {         // if we set "IsInstantiatedOnEachOptimizationIteration" to false (so not using the default of true), the values here would be unexpected         // since the custom trade dictionary was never explicitly reset at the end of each optimization         Print(myTrades.Count);     }   } }   protected override void OnBarUpdate() {   if (CurrentBar < BarsRequiredToTrade)     return;     if (CrossAbove(SMA(Fast), SMA(Slow), 1))   {     EnterLong();     myTrades.Add(Time[0], "long");     }   else if (CrossBelow(SMA(Fast), SMA(Slow), 1))   {     EnterShort();     myTrades.Add(Time[0], "short");   } }   [Range(1, int.MaxValue), NinjaScriptProperty] [Display(Name = "Fast", GroupName = "NinjaScriptStrategyParameters", Order = 0)] public int Fast { get; set; }   [Range(1, int.MaxValue), NinjaScriptProperty] [Display(Name = "Slow", GroupName = "NinjaScriptStrategyParameters", Order = 1)] public int Slow { get; set; } |

| ns **Manually resetting class level variables to take advantage of Strategy Analyzer optimizer performance benefits** | |
| --- | --- |
| // A custom trades dictionary is declared when strategy is first optimized, // but not instantiated until later in State.DataLoaded, private Dictionary<DateTime, string> myTrades;   // examples of other fields which need to be reset private double myDouble; private bool myBool; private DateTime myDateTime; private Order myOrderObject; private Brush myBrushObject; private SMA mySMAIndicator; private Array myIntArray; private List<object> myList; private Series<double> mySeries;   protected override void OnStateChange() {   if (State == State.SetDefaults)   {     Name = "My Optimization Test 2";     Description = "Demonstrates manually resetting a class level variable without re-instantiating the strategy";     Fast = 10;     Slow = 25;           // in this case, we do not need to re-instantiate the strategy after each optimization     // because we are explicitly resetting the custom trade dictionary in State.DataLoaded     // This design of re-using the strategy instance comes with performance benefits     IsInstantiatedOnEachOptimizationIteration = false;   }     else if (State == State.DataLoaded)   {     // re-create custom trade dictionary on each optimization run     // we are guaranteed to start with a new object on each optimization run     if (myTrades != null)       myTrades.Clear();     else       myTrades = new Dictionary<DateTime, string>();           //Any strategy defaults which are maintained do not need to be reset if they are not mutable as the strategy runs.     //Any strategy state that would be mutable after State.SetDefaults needed to be reset for the next run.     myDouble = double.MinValue;     myBool = false;     myDateTime = DateTime.MinValue;     myOrderObject = null;     myBrushObject = null;     mySMAIndicator = SMA(14);           if (myIntArray != null)         Array.Clear(myIntArray, 0, myIntArray.Length);     else         myIntArray = new int[20];           if (myList != null)         myList.Clear();     else         myList = new List<object>();           mySeries = new Series<double>(this);   } }   protected override void OnBarUpdate() {   if (CurrentBar < BarsRequiredToTrade)     return;     if (CrossAbove(SMA(Fast), SMA(Slow), 1))   {     EnterLong();     myTrades.Add(Time[0], "long");     }   else if (CrossBelow(SMA(Fast), SMA(Slow), 1))   {     EnterShort();     myTrades.Add(Time[0], "short");   } }   [Range(1, int.MaxValue), NinjaScriptProperty] [Display(Name = "Fast", GroupName = "NinjaScriptStrategyParameters", Order = 0)] public int Fast { get; set; }   [Range(1, int.MaxValue), NinjaScriptProperty] [Display(Name = "Slow", GroupName = "NinjaScriptStrategyParameters", Order = 1)] public int Slow { get; set; } | |
| **Navigation:**  [NinjaScript](https://ninjatrader.com/es/support/helpGuides/nt8/ninjascript.htm) > [Language Reference](https://ninjatrader.com/es/support/helpGuides/nt8/language_reference_wip.htm) > [Optimizer](https://ninjatrader.com/es/support/helpGuides/nt8/optimizer.htm) >  **OptimizationParameters** | | [Previous page](https://ninjatrader.com/es/support/helpGuides/nt8/onoptimize.htm) [Return to chapter overview](https://ninjatrader.com/es/support/helpGuides/nt8/optimizer.htm) [Next page](https://ninjatrader.com/es/support/helpGuides/nt8/runiteration.htm) |

**Definition**

The optimization parameters selected for the optimization run. (e.g. user parameters or Data Series)

**Property Value**

A bool value.

**Syntax**

Strategies[0].OptimizationParameters

**Examples**

| ns | |
| --- | --- |
| protected override void OnOptimize()  {      // If there are no optimization parameters to optimize, return      if (Strategies[0].OptimizationParameters.Count == 0)          return;        // Do something with the optimization parameter      Parameter parameter = Strategies[0].OptimizationParameters[0];  } | |
| **Navigation:**  [NinjaScript](https://ninjatrader.com/es/support/helpGuides/nt8/ninjascript.htm) > [Language Reference](https://ninjatrader.com/es/support/helpGuides/nt8/language_reference_wip.htm) > [Optimizer](https://ninjatrader.com/es/support/helpGuides/nt8/optimizer.htm) >  **OptimizationParameters** | | [Previous page](https://ninjatrader.com/es/support/helpGuides/nt8/onoptimize.htm) [Return to chapter overview](https://ninjatrader.com/es/support/helpGuides/nt8/optimizer.htm) [Next page](https://ninjatrader.com/es/support/helpGuides/nt8/runiteration.htm) |

**Definition**

The optimization parameters selected for the optimization run. (e.g. user parameters or Data Series)

**Property Value**

A bool value.

**Syntax**

Strategies[0].OptimizationParameters

**Examples**

| ns | |
| --- | --- |
| protected override void OnOptimize()  {      // If there are no optimization parameters to optimize, return      if (Strategies[0].OptimizationParameters.Count == 0)          return;        // Do something with the optimization parameter      Parameter parameter = Strategies[0].OptimizationParameters[0];  } | |
| **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) >  **OptimizationPeriod** | | [Previous page](https://ninjatrader.com/es/support/helpGuides/nt8/onpositionupdate.htm) [Return to chapter overview](https://ninjatrader.com/es/support/helpGuides/nt8/strategy.htm) [Next page](https://ninjatrader.com/es/support/helpGuides/nt8/order.htm) |

**Definition**

Reserved for [Walk-Forward Optimization](https://ninjatrader.com/es/support/helpGuides/nt8/walk_forward_optimize_a_strate.htm), this property determines the number of days used for the "in sample" backtest period for a given strategy.  See also [TestPeriod](https://ninjatrader.com/es/support/helpGuides/nt8/testperiod.htm).

|  |
| --- |
| **Note**:  This property should **ONLY** be called from the [OnStateChange()](https://ninjatrader.com/es/support/helpGuides/nt8/onstatechange.htm) method during State.SetDefaults |

**Property Value**

An int value representing the number of "in sample" days used for walk-forward optimization; Default value is set to 10.

**Syntax**

OptimizationPeriod

**Examples**

| ns | |
| --- | --- |
| protected override void OnStateChange() {     if (State == State.SetDefaults)     {                 //set the default optimization period to 20 days for WFOs         OptimizationPeriod = 20;     } } | |
| **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) >  **OnStateChange()** | | [Previous page](https://ninjatrader.com/es/support/helpGuides/nt8/marketdeptheventargs.htm) [Return to chapter overview](https://ninjatrader.com/es/support/helpGuides/nt8/common.htm) [Next page](https://ninjatrader.com/es/support/helpGuides/nt8/setstate.htm) |

**Definition**

An event driven method which is called whenever the script enters a new [State](https://ninjatrader.com/es/support/helpGuides/nt8/state.htm). The **OnStateChange()** method can be used to configure script properties, create one-time behavior when going from historical to real-time, as well as manage clean up resources on termination.

|  |
| --- |
| **Notes**:  •Viewing any UI element which lists NinjaScript classes (such as the Indicators or Strategies window, a chart's Chart Style dropdown menu, etc.) will initialize all classes of that Type when it is opened, which causes each script to enter **State.SetDefaults**, even if it is not actively configured or running in any window. It is important to keep this in mind when adding logic within **State.SetDefaults** in **OnStateChange()**, as this logic will be processed each time the script is initialized. For example, opening the Indicators window will trigger **State.SetDefaults** for all indicators in the system, and closing the Indicators window will trigger**State.Terminated** for all Indicators. In addition, disconnecting or connecting to a data provider can cause State transitions for any currently active scripts. Further discussion of this aspect of the state change model can be found via [*Understanding the lifecycle of your NinjaScript objects*](https://ninjatrader.com/es/support/helpGuides/nt8/understanding_the_lifecycle_of.htm).  •When an indicator is configured on a chart while a Compile is taking place in the NinjaScript Editor, it can appear that the script passes through **State.Terminated**. However, this is the result of a copy of the script being initialized at compile-time, NOT the result of the indicator on the chart being disabled and re-initialized. |

**Related Methods and Properties**

|  |  |
| --- | --- |
| [SetState()](https://ninjatrader.com/es/support/helpGuides/nt8/setstate.htm) | Method is used for changing the State of any running NinjaScript object. |
| [State](https://ninjatrader.com/es/support/helpGuides/nt8/state.htm) | Represents the current progression of the object as it advances from setup, processing data, to termination. |

**Method Return Value**

This method does not return a value.

**Syntax**  
See example below. The NinjaScript wizards automatically generate the method syntax for you.

Possible states are:

|  |  |  |
| --- | --- | --- |
| **State Name** | **This state is called when** | **This state is where you should** |
| State.SetDefaults | **SetDefaults** is always called when displaying objects in a UI list such as the Indicators dialogue window since temporary objects are created for the purpose of UI display | •Keep as lean as possible  •Set default values (pushed to UI) |
| State.Configure | **Configure** is called after a user adds an object to the applied list of objects and presses the OK or Apply button.  This state is called only once for the life of the object. | •Add additional data series via [AddDataSeries()](https://ninjatrader.com/es/support/helpGuides/nt8/adddataseries.htm)  •Declare custom resources |
| State.Active | **Active** is called once after the object is configured and is ready to process data (DrawingTools could see multiple calls as internally an object for hit testing is cloned) | •Used for objects such as [Share Service](https://ninjatrader.com/es/support/helpGuides/nt8/share_service.htm) which do not process price series data  •Indicate the object is ready to being processing information |
| State.DataLoaded | **DataLoaded** is called only once after all data series have been loaded. | •Use for logic that needs to access data related objects like Bars, Instruments, BarsPeriod, TradingHours or instantiating indicators  •Notify that all data series have been loaded  •Initialize any class level variables (including custom [Series<T>](https://ninjatrader.com/es/support/helpGuides/nt8/seriest.htm) objects) |
| State.Historical | **Historical**is called once the object begins to process historical data. This state is called once when running an object in real-time. This object is called multiple times when running a backtest optimization and the property [IsInstantiatedOnEachOptimizationIteration](https://ninjatrader.com/es/support/helpGuides/nt8/isinstantiatedoneachoptimizationiteration.htm) is true (default behavior) | •Notify that the object is processing historical data |
| State.Transition | **Transition** is called once as the object has finished processing historical data but before it starts to process realtime data. | •Notify that the indicator or strategy is is transitioning to realtime data  •Prepare realtime related resources |
| State.Realtime | **Realtime** is called once when the object begins to process realtime data. | •Notify that the indicator or strategy is processing realtime data  •Execute realtime related logic |
| State.Terminated | **Terminated** is called once when the object terminates. | •Notify the object is shutting down  •Use to clean up/dispose of resources |

**Active States vs Data Processing States**

After **State.Configure,**each type of NinjaScript type has its own state management system which can be classified under two categories:

•**Active state:**  State.Active

•**Data Processing states:**State.DataLoaded, State.Historical, State.Transition, State.Realtime

The table below lists each NinjaScript type and it's designed state management system:

|  |  |
| --- | --- |
| **NinjaScript Type** | **State Management System** |
| AddOns\* | Active state |
| BarTypes | Active state |
| ChartStyles | Active state |
| DrawingTools | Active state |
| Indicators | Data Processing states |
| ImportTypes | Active state |
| Market Analyzer Columns | Data Processing states |
| OptimizationFitnesses | Active state |
| Optimizers | Active state |
| PerformanceMetrics | Active state |
| ShareServices | Active state |
| Strategies | Data Processing states |
| SuperDOM Columns | Active state |

|  |
| --- |
| **Tips:**  •Resources created in **State.Configure** and not disposed of immediately will be kept and utilized if the NinjaScript object resides in grids (e.g. Strategy tab on Control Center), even if it is not enabled. Try to create resources in**State.Historical** or **State.DataLoaded** instead, if possible.  •**State.Historical** is called multiple times when running a backtest [optimization](https://ninjatrader.com/es/support/helpGuides/nt8/optimize_a_strategy.htm) on a strategy and the property "[IsInstantiatedOnEachOptimizationIteration](https://ninjatrader.com/es/support/helpGuides/nt8/isinstantiatedoneachoptimizationiteration.htm)" is **true** (default behavior).  • Scripts that require [Calculate](https://ninjatrader.com/es/support/helpGuides/nt8/calculate.htm) to be set by the developer must set this property in **State.Historical** in order to ensure that if this script is a child (hosted) that the parent.Calculate property which is adopted by the child is overridden again.  •When instantiating indicators in a [Multi-Series script](https://ninjatrader.com/es/support/helpGuides/nt8/multi-time_frame__instruments.htm) in OnStateChange, 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) |

**Examples**

| ns | |
| --- | --- |
| protected override void OnStateChange() {   if (State == State.SetDefaults)   {     // Calculate once at the end of every single bar     Calculate = Calculate.OnBarClose;         // Add two plots     AddPlot(Brushes.Blue, "Upper"));     AddPlot(Brushes.Orange, "Lower"));   }     else if (State == State.Configure)   {     // Adds a 5-minute Bars object to the strategy and is automatically assigned     // a Bars object index of 1 since the primary data the strategy is run against     // set by the UI takes the index of 0.             AddDataSeries("AAPL", BarsPeriodType.Minute, 5);       } } | |
| **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) >  **Account** | | [Previous page](https://ninjatrader.com/es/support/helpGuides/nt8/strategy.htm) [Return to chapter overview](https://ninjatrader.com/es/support/helpGuides/nt8/strategy.htm) [Next page](https://ninjatrader.com/es/support/helpGuides/nt8/addchartindicator.htm) |

**Definition**

Represents the real-world or simulation **Account** configured for the strategy.

**Property Value**

An [Account](https://ninjatrader.com/es/support/helpGuides/nt8/account_class.htm) object configured for the strategy

**Syntax**

Account

**Examples**

| ns | |
| --- | --- |
| //Displays text on chart indicating what account the strategy is applied to  Draw.TextFixed(this, "tag1", "Strategy is applied to " + Account.Name, TextPosition.BottomRight); | |
| **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) >  **AddChartIndicator()** | | [Previous page](https://ninjatrader.com/es/support/helpGuides/nt8/strategy_account.htm) [Return to chapter overview](https://ninjatrader.com/es/support/helpGuides/nt8/strategy.htm) [Next page](https://ninjatrader.com/es/support/helpGuides/nt8/addperformancemetric.htm) |

**Definition**

Adds an indicator to the strategy only for the purpose of displaying it on a chart.

|  |
| --- |
| **Notes**:  •Only the Plot properties of an indicator added by AddChartIndicator() will be accessible in the Indicators dialogue on charts. Other properties must be set in code.  •To add Bars objects to your strategy for calculation purposes see the [AddDataSeries()](https://ninjatrader.com/es/support/helpGuides/nt8/adddataseries.htm) method.  •An indicator being added via AddChartIndicator() cannot use any additional data series hosted by the calling strategy, but can only use the strategy's primary data series. If you wish to use a different data series for the indicator's input, you can add the series in the indicator itself and explicitly reference it in the indicator code (please make sure though the hosting strategy has the same [AddDataSeries()](https://ninjatrader.com/es/support/helpGuides/nt8/adddataseries.htm) call included as well)  o If a secondary or null Bars series is specified by the calling strategy (not the indicator itself), the strategy's primary series will be substituted instead.  •Dynamically using [DrawOnPricePanel](https://ninjatrader.com/es/support/helpGuides/nt8/drawonpricepanel.htm) in an indicator outside of State.SetDefaults may show issues when working with that indicator through a hosting strategy via [AddChartIndicator()](https://ninjatrader.com/es/support/helpGuides/nt8/addchartindicator.htm). |

**Method Return Value**

This method does not return a value.

**Syntax**  
AddChartIndicator(IndicatorBase indicator)

|  |
| --- |
| **Warning**:  This method should **ONLY** be called from the [OnStateChange()](https://ninjatrader.com/es/support/helpGuides/nt8/onstatechange.htm) method during **State.DataLoaded** |

**Parameters**

|  |  |
| --- | --- |
| indicator | An indicator object |

**Examples**

| ns |
| --- |
| protected override void OnStateChange() {     if (State == State.DataLoaded)     {         // Charts a 20 period simple moving average to the chart         AddChartIndicator(SMA(20));     } } |

|  |
| --- |
| **Tip**:  If you are adding an indicator which is dependent on the correct [State](https://ninjatrader.com/es/support/helpGuides/nt8/state.htm) of the indicator, you will need to ensure that you are also calling the indicator from the strategy in [OnBarUpdate()](https://ninjatrader.com/es/support/helpGuides/nt8/onbarupdate.htm), otherwise your indicator will only process in **State.RealTime** for performance optimizations. |

| ns | |
| --- | --- |
| protected override void OnStateChange() {   if (State == State.DataLoaded)   {     // Charts a 20 period simple moving average to the chart     AddChartIndicator(SMA(20));   } }   protected override void OnBarUpdate() {     // call SMA() historically to ensure the indicator processes its historical states as well   double sma = SMA(20)[0]; } | |
| **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) >  **AddPerformanceMetric()** | | [Previous page](https://ninjatrader.com/es/support/helpGuides/nt8/addchartindicator.htm) [Return to chapter overview](https://ninjatrader.com/es/support/helpGuides/nt8/strategy.htm) [Next page](https://ninjatrader.com/es/support/helpGuides/nt8/atm_strategy_methods.htm) |

**Definition**

Adds an instance of custom [Performance Metric](https://ninjatrader.com/es/support/helpGuides/nt8/performancemetrics.htm) to a strategy used in strategy calculations.

**Method Return Value**

This method does not return a value.

**Syntax**  
AddPerformanceMetric(PerformanceMetricBase performanceMetric)

|  |
| --- |
| **Warning**:  This method should **ONLY** be called from the [OnStateChange()](https://ninjatrader.com/es/support/helpGuides/nt8/onstatechange.htm) method during **State.Configure** |

**Parameters**

|  |  |
| --- | --- |
| performanceMetric | The performance metric object to be added |

**Examples**

| ns |
| --- |
| protected override void OnStateChange() {     if (State == State.Configure)     {         AddPerformanceMetric(new NinjaTrader.NinjaScript.PerformanceMetrics.SampleCumProfit());     } } |

|  |  |
| --- | --- |
| **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) >  **AddPerformanceMetric()** | [Previous page](https://ninjatrader.com/es/support/helpGuides/nt8/addchartindicator.htm) [Return to chapter overview](https://ninjatrader.com/es/support/helpGuides/nt8/strategy.htm) [Next page](https://ninjatrader.com/es/support/helpGuides/nt8/atm_strategy_methods.htm) |

**Definition**

Adds an instance of custom [Performance Metric](https://ninjatrader.com/es/support/helpGuides/nt8/performancemetrics.htm) to a strategy used in strategy calculations.

**Method Return Value**

This method does not return a value.

**Syntax**  
AddPerformanceMetric(PerformanceMetricBase performanceMetric)

|  |
| --- |
| **Warning**:  This method should **ONLY** be called from the [OnStateChange()](https://ninjatrader.com/es/support/helpGuides/nt8/onstatechange.htm) method during **State.Configure** |

**Parameters**

|  |  |
| --- | --- |
| performanceMetric | The performance metric object to be added |

**Examples**

| ns |
| --- |
| protected override void OnStateChange() {     if (State == State.Configure)     {         AddPerformanceMetric(new NinjaTrader.NinjaScript.PerformanceMetrics.SampleCumProfit());     } } |

|  |  |
| --- | --- |
| **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) > [ATM Strategy Methods](https://ninjatrader.com/es/support/helpGuides/nt8/atm_strategy_methods.htm) >  **AtmStrategyChangeEntryOrder()** | [Previous page](https://ninjatrader.com/es/support/helpGuides/nt8/atmstrategycancelentryorder.htm) [Return to chapter overview](https://ninjatrader.com/es/support/helpGuides/nt8/atm_strategy_methods.htm) [Next page](https://ninjatrader.com/es/support/helpGuides/nt8/atmstrategychangestoptarget.htm) |

**Definition**

Changes the price of the specified entry order.

**Method Return Value**

Returns **true** if the specified order was found; otherwise **false**.

**Syntax**

AtmStrategyChangeEntryOrder(double limitPrice, double stopPrice, string orderId)

**Parameters**

|  |  |
| --- | --- |
| limitPrice | Order limit price |
| stopPrice | Order stop price |
| orderId | The unique identifier for the entry order |

**Examples**

| ns | |
| --- | --- |
| protected override void OnBarUpdate() {     AtmStrategyChangeEntryOrder(GetCurrentBid(), 0, "orderIdValue"); } | |
| **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) > [ATM Strategy Methods](https://ninjatrader.com/es/support/helpGuides/nt8/atm_strategy_methods.htm) >  **AtmStrategyChangeStopTarget()** | | [Previous page](https://ninjatrader.com/es/support/helpGuides/nt8/atmstrategychangeentryorder.htm) [Return to chapter overview](https://ninjatrader.com/es/support/helpGuides/nt8/atm_strategy_methods.htm) [Next page](https://ninjatrader.com/es/support/helpGuides/nt8/atmstrategyclose.htm) |

**Definition**

Changes the price of the specified order of the specified ATM strategy.

**Method Return Value**

Returns **true** if the specified order was found; otherwise **false**.

**Syntax**

AtmStrategyChangeStopTarget(double *limitPrice*, double *stopPrice*, string *orderName*, string a*tmStrategyId*)

**Parameters**

|  |  |
| --- | --- |
| limitPrice | Order limit price |
| stopPrice | Order stop price |
| orderName | The order name such as "Stop1" or "Target2" |
| atmStrategyId | The unique identifier for the ATM strategy |

**Examples**

| ns | |
| --- | --- |
| protected override void OnBarUpdate() {     AtmStrategyChangeStopTarget(0, SMA(10)[0], "Stop1", "AtmIdValue"); } | |
| **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) > [ATM Strategy Methods](https://ninjatrader.com/es/support/helpGuides/nt8/atm_strategy_methods.htm) >  **AtmStrategyClose()** | | [Previous page](https://ninjatrader.com/es/support/helpGuides/nt8/atmstrategychangestoptarget.htm) [Return to chapter overview](https://ninjatrader.com/es/support/helpGuides/nt8/atm_strategy_methods.htm) [Next page](https://ninjatrader.com/es/support/helpGuides/nt8/atmstrategycreate.htm) |

**Definition**

Cancels any working orders and closes any open position of a strategy using the default [ATM strategy close behavior](https://ninjatrader.com/es/support/helpGuides/nt8/closing_a_position_or_atm_stra.htm).

**Method Return Value**

Returns **true** if the specified ATM strategy was found; otherwise **false**.

|  |
| --- |
| **Note**:  A method return value of **true** in **NO WAY** indicates that the strategy in fact is closed. It indicates that the the specified ATM strategy was found and the internal close routine was triggered. |

**Syntax**

AtmStrategyClose(string a*tmStrategyId*)

**Parameters**

|  |  |
| --- | --- |
| atmStrategyId | The unique identifier for the ATM strategy |

**Examples**

| ns | |
| --- | --- |
| protected override void OnBarUpdate() {     // Check for valid condition and create an ATM Strategy     if (GetAtmStrategyUnrealizedProfitLoss("idValue") > 500)         AtmStrategyClose("idValue"); } | |
| **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) > [ATM Strategy Methods](https://ninjatrader.com/es/support/helpGuides/nt8/atm_strategy_methods.htm) >  **AtmStrategyCreate()** | | [Previous page](https://ninjatrader.com/es/support/helpGuides/nt8/atmstrategyclose.htm) [Return to chapter overview](https://ninjatrader.com/es/support/helpGuides/nt8/atm_strategy_methods.htm) [Next page](https://ninjatrader.com/es/support/helpGuides/nt8/getatmstrategyentryorderstatus.htm) |

**Definition**

Submits an entry order that will execute a specified ATM Strategy.

|  |
| --- |
| **Notes**:    •Please review the section on using [ATM Strategies](https://ninjatrader.com/es/support/helpGuides/nt8/using_atm_strategies.htm)  •This method is **NOT** backtestable and will **NOT** execute on historical data  •See the [AtmStrategyCancelEntryOrder()](https://ninjatrader.com/es/support/helpGuides/nt8/atmstrategycancelentryorder.htm) to cancel an entry order  •See the [AtmStrategyChangeEntryOrder()](https://ninjatrader.com/es/support/helpGuides/nt8/atmstrategychangeentryorder.htm) to change the price of the entry order  •The ATM Strategy will be created on the Hosting NinjaScripts UI Thread. A callback is provided to check when the ATM Strategy is started on that thread.  •Please see the SampleATMStrategy build into NinjaTrader for example usage. |

**Method Return Value**

This method does not return a value

**Syntax**

AtmStrategyCreate(OrderAction *action*, OrderType *orderType*, double *limitPrice*, double *stopPrice*, TimeInForce *timeInForce*, string *orderId*, string *strategyTemplateName*, string a*tmStrategyId, Action<Cbi.ErrorCode, string> callback)*

**Parameters**

|  |  |
| --- | --- |
| action | Sets if the entry order is a buy or sell order    Possible values are:    •OrderAction.Buy  •OrderAction.Sell |
| orderType | Sets the order type of the entry order    Possible values are:    •OrderType.Limit  •OrderType.Market  •OrderType.MIT  •OrderType.StopMarket  •OrderType.StopLimit |
| limitPrice | The limit price of the order |
| stopPrice | The stop price of the order |
| timeInForce | Sets the time in force of the entry order    Possible values are:  •TimeInForce.Day  •TimeInForce.Gtc |
| orderId | The unique identifier for the entry order |
| strategyTemplateName | Specifies which strategy template will be used |
| atmStrategyId | The unique identifier for the ATM strategy |
| callback | The callback action is used to check that the ATM Strategy is successfully started |

|  |
| --- |
| **Tip**:  Unlike NinjaScript Strategy orders (both [managed](https://ninjatrader.com/es/support/helpGuides/nt8/managed_approach.htm) and [unmanaged](https://ninjatrader.com/es/support/helpGuides/nt8/unmanaged_approach.htm)), ATM strategies generated by the AtmStrategyCreate() method can then be managed manually by any order entry window such as the SuperDOM or within your NinjaScript strategy. |

**Examples**

| ns | |
| --- | --- |
| private string atmStrategyId; private string atmStrategyOrderId; private bool   isAtmStrategyCreated = false;   protected override void OnBarUpdate() {   if (State < State.Realtime)       return;     if (Close[0] > SMA(20)[0])   {       atmStrategyId = GetAtmStrategyUniqueId();       atmStrategyOrderId = GetAtmStrategyUniqueId();         AtmStrategyCreate(OrderAction.Buy, OrderType.Market, 0, 0, TimeInForce.Day,           atmStrategyOrderId, "MyTemplate", atmStrategyId, (atmCallbackErrorCode, atmCallbackId) => {             // checks that the call back is returned for the current atmStrategyId stored           if (atmCallbackId == atmStrategyId)           {               // check the atm call back for any error codes               if (atmCallbackErrorCode == Cbi.ErrorCode.NoError)               {                   // if no error, set private bool to true to indicate the atm strategy is created                   isAtmStrategyCreated = true;               }           }       });   }     if(isAtmStrategyCreated)   {       // atm logic   }     else if(!isAtmStrategyCreated)   {       // custom handling for a failed atm Strategy   } } | |
| **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** | | [Previous page](https://ninjatrader.com/es/support/helpGuides/nt8/signature.htm) [Return to chapter overview](https://ninjatrader.com/es/support/helpGuides/nt8/language_reference_wip.htm) [Next page](https://ninjatrader.com/es/support/helpGuides/nt8/strategy_account.htm) |

The methods and properties covered in this section are unique to custom strategy development.

**In this section**

|  |  |
| --- | --- |
| [Account](https://ninjatrader.com/es/support/helpGuides/nt8/strategy_account.htm) | Represents the real-world or simulation **Account** configured for the strategy. |
| [AddChartIndicator()](https://ninjatrader.com/es/support/helpGuides/nt8/addchartindicator.htm) | Adds an indicator to the strategy only for the purpose of displaying it on a chart. |
| [AddPerformanceMetric()](https://ninjatrader.com/es/support/helpGuides/nt8/addperformancemetric.htm) | Adds an instance of custom [Performance Metric](https://ninjatrader.com/es/support/helpGuides/nt8/performancemetrics.htm) to a strategy used in strategy calculations. |
| [ATM Strategy Methods](https://ninjatrader.com/es/support/helpGuides/nt8/atm_strategy_methods.htm) | Adds ATM strategies to manage your position |
| [BarsRequiredToTrade](https://ninjatrader.com/es/support/helpGuides/nt8/barsrequiredtotrade.htm) | The number of historical bars required before the strategy starts processing order methods called in the [OnBarUpdate()](https://ninjatrader.com/es/support/helpGuides/nt8/onbarupdate.htm) method. |
| [BarsSinceEntryExecution()](https://ninjatrader.com/es/support/helpGuides/nt8/barssinceentryexecution.htm) | Returns the number of bars that have elapsed since the last specified entry. |
| [BarsSinceExitExecution()](https://ninjatrader.com/es/support/helpGuides/nt8/barssinceexitexecution.htm) | Returns the number of bars that have elapsed since the last specified exit. |
| [ChartIndicators](https://ninjatrader.com/es/support/helpGuides/nt8/chartindicators.htm) | Contains a collection of Indicators which have been added to the strategy instance using [AddChartIndicator()](https://ninjatrader.com/es/support/helpGuides/nt8/addchartindicator.htm). |
| [CloseStrategy()](https://ninjatrader.com/es/support/helpGuides/nt8/closestrategy.htm) | Cancels all working orders, closes any existing positions, and finally disables the strategy. |
| [ConnectionLossHandling](https://ninjatrader.com/es/support/helpGuides/nt8/connectionlosshandling.htm) | Sets the manner in which your strategy will behave when a connection loss is detected. |
| [DaysToLoad](https://ninjatrader.com/es/support/helpGuides/nt8/daystoload.htm) | Determines the number of trading days which will be configured when loading the strategy from the **Strategies Grid**. |
| [DefaultQuantity](https://ninjatrader.com/es/support/helpGuides/nt8/defaultquantity.htm) | An order size variable that can be set either programmatically or overriden via the Strategy that determines the quantity of an entry order. |
| [DisconnectDelaySeconds](https://ninjatrader.com/es/support/helpGuides/nt8/disconnectdelayseconds.htm) | Determines the amount of time a disconnect would have to last before [connection loss handling](https://ninjatrader.com/es/support/helpGuides/nt8/connectionlosshandling.htm) takes action. |
| [EntriesPerDirection](https://ninjatrader.com/es/support/helpGuides/nt8/entriesperdirection.htm) | Determines the maximum number of entries allowed per direction while a position is active based on the [EntryHandling](https://ninjatrader.com/es/support/helpGuides/nt8/entryhandling.htm) property. |
| [EntryHandling](https://ninjatrader.com/es/support/helpGuides/nt8/entryhandling.htm) | Sets the manner in how entry orders will handle. |
| [Execution](https://ninjatrader.com/es/support/helpGuides/nt8/execution.htm) | Represents a read only interface that exposes information regarding an execution (filled order) resulting from an order and is passed as a parameter in the [OnExecutionUpdate()](https://ninjatrader.com/es/support/helpGuides/nt8/onexecutionupdate.htm) method. |
| [ExitOnSessionCloseSeconds](https://ninjatrader.com/es/support/helpGuides/nt8/exitonsessioncloseseconds.htm) | The number of seconds before the actual session end time that the "[IsExitOnSessionCloseStrategy](https://ninjatrader.com/es/support/helpGuides/nt8/isexitonsessionclosestrategy.htm)" function will trigger. |
| [IncludeCommission](https://ninjatrader.com/es/support/helpGuides/nt8/includecommission.htm) | Determines if the strategy performance results will include commission on a historical backtest. |
| [IncludeTradeHistoryInBacktest](https://ninjatrader.com/es/support/helpGuides/nt8/includetradehistoryinbacktest.htm) | Determines if the strategy will save orders, trades, and execution history. |
| [IsAdoptAccountPositionAware](https://ninjatrader.com/es/support/helpGuides/nt8/isadoptaccountpositionaware.htm) | Determines if the strategy is programmed in a manner capable of handling  real-world account positions. |
| [IsExitOnSessionCloseStrategy](https://ninjatrader.com/es/support/helpGuides/nt8/isexitonsessionclosestrategy.htm) | Determines if the strategy will cancel all strategy generated orders and close all open strategy positions at the close of the session. |
| [IsFillLimitOnTouch](https://ninjatrader.com/es/support/helpGuides/nt8/isfilllimitontouch.htm) | Determines if the strategy will use a more liberal fill algorithm for back-testing purposes only. |
| [IsInstantiatedOnEachOptimizationIteration](https://ninjatrader.com/es/support/helpGuides/nt8/isinstantiatedoneachoptimizationiteration.htm) | Determines if the strategy should be re-instantiated (re-created) after each optimization run when using the [Strategy Analyzer Optimizer](https://ninjatrader.com/es/support/helpGuides/nt8/optimize_a_strategy.htm). |
| [IsTradingHoursBreakLineVisible](https://ninjatrader.com/es/support/helpGuides/nt8/istradinghoursbreaklinevisible.htm) | Plots trading hours break lines on the indicator panel. |
| [IsWaitUntilFlat](https://ninjatrader.com/es/support/helpGuides/nt8/iswaituntilflat.htm) | Indicates the strategy is currently waiting until a flat position is detected before submitting live orders. |
| [NumberRestartAttempts](https://ninjatrader.com/es/support/helpGuides/nt8/numberrestartattempts.htm) | Determines the maximum number of restart attempts allowed within the last x minutes defined in [RestartsWithinMinutes](https://ninjatrader.com/es/support/helpGuides/nt8/restartswithinminutes.htm) when the strategy experiences a connection loss. |
| [OnAccountItemUpdate()](https://ninjatrader.com/es/support/helpGuides/nt8/onaccountitemupdate.htm) | An event driven method used for strategies which is called for each AccountItem update for the account on which the strategy is running. |
| [OnExecutionUpdate()](https://ninjatrader.com/es/support/helpGuides/nt8/onexecutionupdate.htm) | An event driven method which is called on an incoming execution of an order managed by a strategy. |
| [OnOrderTrace()](https://ninjatrader.com/es/support/helpGuides/nt8/onordertrace.htm) | An event driven method used for strategies which will allow you to customize the output of [TraceOrders](https://ninjatrader.com/es/support/helpGuides/nt8/traceorders.htm). |
| [OnOrderUpdate()](https://ninjatrader.com/es/support/helpGuides/nt8/onorderupdate.htm) | An event driven method which is called each time an order managed by a strategy changes state. |
| [OnPositionUpdate()](https://ninjatrader.com/es/support/helpGuides/nt8/onpositionupdate.htm) | An event driven method which is called each time the position of a strategy changes state. |
| [OptimizationPeriod](https://ninjatrader.com/es/support/helpGuides/nt8/order.htm) | Reserved for [Walk-Forward Optimization](https://ninjatrader.com/es/support/helpGuides/nt8/walk_forward_optimize_a_strate.htm), this property determines the number of days used for the "in sample" backtest period for a given strategy.  See also [TestPeriod](https://ninjatrader.com/es/support/helpGuides/nt8/testperiod.htm). |
| [Order](https://ninjatrader.com/es/support/helpGuides/nt8/order.htm) | Represents a read only interface that exposes information regarding an order. |
| [Order Methods](https://ninjatrader.com/es/support/helpGuides/nt8/order_methods.htm) | NinjaScript provides several approaches you can use for order placement within your NinjaScript strategy. |
| [OrderFillResolution](https://ninjatrader.com/es/support/helpGuides/nt8/orderfillresolution.htm) | Determines how strategy orders are filled during historical states. |
| [OrderFillResolutionType](https://ninjatrader.com/es/support/helpGuides/nt8/orderfillresolutiontype.htm) | Determines the bars type which will be used for historical fill processing. |
| [OrderFillResolutionValue](https://ninjatrader.com/es/support/helpGuides/nt8/orderfillresolutionvalue.htm) | Determines the bars period interval value which will be used for historical fill processing. |
| [PerformanceMetrics](https://ninjatrader.com/es/support/helpGuides/nt8/strategy_performancemetrics.htm) | Holds an array of [PerformanceMetrics](https://ninjatrader.com/es/support/helpGuides/nt8/performancemetrics.htm) objects that represent custom metrics that can be used for strategy calcuations. |
| [Plots](https://ninjatrader.com/es/support/helpGuides/nt8/strategy_plots.htm) | A collection holding all of the Plot objects that define their visualization characteristics. |
| [Position](https://ninjatrader.com/es/support/helpGuides/nt8/position.htm) | Represents position related information that pertains to an instance of a strategy. |
| [PositionAccount](https://ninjatrader.com/es/support/helpGuides/nt8/positionaccount.htm) | Represents position related information that pertains to real-world account (live or simulation). |
| [Positions](https://ninjatrader.com/es/support/helpGuides/nt8/positions.htm) | Holds an array of [Position](https://ninjatrader.com/es/support/helpGuides/nt8/position.htm) objects that represent positions managed by the strategy. |
| [PositionsAccount](https://ninjatrader.com/es/support/helpGuides/nt8/positionsaccount.htm) | Holds an array of [PositionAccount](https://ninjatrader.com/es/support/helpGuides/nt8/positionaccount.htm) objects that represent positions managed by the strategy's account. |
| [RealtimeErrorHandling](https://ninjatrader.com/es/support/helpGuides/nt8/realtimeerrorhandling.htm) | Defines the behavior of a strategy when a strategy generated order is returned from the broker's server in a "Rejected" state. |
| [RestartsWithinMinutes](https://ninjatrader.com/es/support/helpGuides/nt8/restartswithinminutes.htm) | Determines within how many minutes the strategy will attempt to restart. |
| [SetOrderQuantity](https://ninjatrader.com/es/support/helpGuides/nt8/setorderquantity.htm) | Determines how order sizes are calculated for a given strategy. |
| [Slippage](https://ninjatrader.com/es/support/helpGuides/nt8/slippage.htm) | Sets the amount of slippage in ticks per execution used in performance calculations during backtests. |
| [StartBehavior](https://ninjatrader.com/es/support/helpGuides/nt8/startbehavior.htm) | Sets the start behavior of the strategy. See [Syncing Account Positions](https://ninjatrader.com/es/support/helpGuides/nt8/syncing_account_positions.htm) for more information. |
| [StopTargetHandling](https://ninjatrader.com/es/support/helpGuides/nt8/stoptargethandling.htm) | Determines how stop and target orders are submitted during an entry order execution. |
| [StrategyBaseConverter](https://ninjatrader.com/es/support/helpGuides/nt8/strategybaseconverter.htm) | A custom TypeConverter class handling the designed behavior of an strategy's property descriptor collection. |
| [SystemPerformance](https://ninjatrader.com/es/support/helpGuides/nt8/systemperformance.htm) | The SystemPerformance object holds all trades and trade performance data generated by a strategy. |
| [TestPeriod](https://ninjatrader.com/es/support/helpGuides/nt8/testperiod.htm) | Reserved for [Walk-Forward Optimization](https://ninjatrader.com/es/support/helpGuides/nt8/walk_forward_optimize_a_strate.htm), this property determines the number of days used for the "out of sample" backtest period for a given strategy. |
| [TimeInForce](https://ninjatrader.com/es/support/helpGuides/nt8/timeinforce.htm) | Sets the time in force property for all orders generated by a strategy. |
| [TraceOrders](https://ninjatrader.com/es/support/helpGuides/nt8/traceorders.htm) | Determines if OnOrderTrace() would be called for a given strategy. |
| [Trade](https://ninjatrader.com/es/support/helpGuides/nt8/trade.htm) | A Trade is a completed buy/sell or sell/buy transaction. It consists of an entry and exit execution. |
| [TradeCollection](https://ninjatrader.com/es/support/helpGuides/nt8/tradecollection.htm) | A collection of [Trade](https://ninjatrader.com/es/support/helpGuides/nt8/trade.htm) objects. |
| [TradesPerformanceValues](https://ninjatrader.com/es/support/helpGuides/nt8/tradesperformancevalues.htm) | Performance values of a [collection](https://ninjatrader.com/es/support/helpGuides/nt8/tradecollection.htm) of [Trade](https://ninjatrader.com/es/support/helpGuides/nt8/trade.htm) objects. |
| [WaitForOcoClosingBracket](https://ninjatrader.com/es/support/helpGuides/nt8/waitforococlosingbracket.htm) | Determines if the strategy will submit both legs of an OCO bracket before submitting the pair to the broker. |
| **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#MiscellaneousPractices)

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

The following are what we suggest for best practices for distribution.

**Do not deploy NinjaScript Source Files**

If you are a commercial vendor, you should never distribute the NinjaScript .cs source code files even if your IP is contained within an assembly or proprietary DLL. Source code files are editable by users and can result in unnecessary support issues.

**Naming Conventions**

Please use consistent naming convention with your indicators and strategies. We suggest adding a prefix to an indicator name. If your company name is "Hyper" you could name your indicators "HyperTrend" or "HyperOscillator" for example.

In the event that you provide NinjaScript export archives (zip files) as your means of distribution, NinjaTrader will automatically block incompatible scripts from importing so there will be no confusion by the user as to whether they are installing Version 7 or 8 scripts to their NinjaTrader installation. It is advisable to include the NinjaTrader version number in the export archive which will reduce potential support burden. For example, you could name your indicators “MyIndicator\_7.zip” and “MyIndicator\_8.zip”.

**Clean up your resources**

Always free up resources such as external windows DLL's or license management related resources. Resources should be freed within the [OnStateChange()](https://ninjatrader.com/es/support/helpGuides/nt8/onstatechange.htm) method in State.Terminate. NinjaTrader calls this method at the point at which a script is no longer used.

**User Authentication Trigger**

If you use a proprietary user authentication process, ensure that it is triggered within the [OnStateChange()](https://ninjatrader.com/es/support/helpGuides/nt8/onstatechange.htm) method in State.SetDefaults. This ensures that users are not forced to endure unnecessary delays on NinjaTrader start up or dialog windows that display available indicators and strategies as the windows are loaded. NinjaTrader, LLC provides a free licensing service for qualified 3rd party developers. For more information on this free service, contact your NinjaTrader Business Development representative.

**User Authentication Check State**

A license check should only be performed once and maintain its check state.

**User Authentication Time Out**

A license check should have a time out in case of internet issues, to enhance performance in this case.

**Custom Installer**

If you provide a custom installer, the installer should not overwrite any NinjaTrader deployed files, and you should provide an uninstall option which removes all installed files.

It is also preferred that you provide one installer that provides the user the option to install either a version 7 or version 8 compatible version of your product(s). Ensure that you only copy the correct files to the correct NinjaTrader installation folders since if you don’t it is possible that it could cause compile issues for the customer and it will be extremely difficult for all involved to isolate the cause.

These are the following folder names:

•Documents\NinjaTrader 7\bin\Custom

•Documents\NinjaTrader 8\bin\Custom

**Test on Legacy Operating Systems**

SomeNinjaTrader customers run on older Operating Systems (such as Windows 7) and you should make sure that your indicators, custom installers and external DLLs (if any are used) properly run on these legacy operating systems.

**Expose Indicator States**

If your proprietary indicator acts as a trend state (green bars are bullish, red bearish) its good practice to expose the indicators's state so that consumers of your indicators can use them within their own custom indicator or strategy.

|  |  |
| --- | --- |
| **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" \l "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" \l "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:**  »No topics above this level«  **Welcome to NinjaTrader 8 Beta** | [Return to chapter overview](https://ninjatrader.com/es/support/helpGuides/nt8/sharpdx_directwrite_textlayout.htm) |

**Welcome to the NinjaTrader 8 Beta Release!**

We are very excited to openly release NinjaTrader 8 Beta to our general user base. The purpose of this program is to have you start testing the platform to in order to solicit feedback in the form of bug reports and constructive criticism. The beta process is a critical step in the software development cycle during which time we jointly work towards improving quality and performance with a goal of reaching a stable product ready for general release.

This release marks the introduction of our next generation trading platform. NinjaTrader 8 adds over 500+ new features and functionality on top of a re-designed architecture built to take advantage of new development practices and modern technologies. This is our most ambitious project to date and we are excited to get it in the hands of our customers.

|  |  |
| --- | --- |
| **NinjaTrader 8 Download** |  |
| [8.0.0.14](http://www.ninjatrader.com/PlatformDirect) | [Release Notes](https://ninjatrader.com/es/support/helpGuides/nt8/8_0_0_14.htm) |
| [8.0.0.13](http://www.ninjatrader.com/ninjatrader/nt8/NinjaTrader.Install.B13.msi) | [Release Notes](https://ninjatrader.com/es/support/helpGuides/nt8/8_0_0_13.htm) |
| [8.0.0.12](http://www.ninjatrader.com/ninjatrader/nt8/NinjaTrader.Install.B12.msi) | [Release Notes](https://ninjatrader.com/es/support/helpGuides/nt8/8_0_0_12.htm) |

**Bug reports and providing feedback**

Support will be driven via our[NinjaTrader Support Forum](http://www.ninjatrader.com/support/forum/forumdisplay.php?f=57) NinjaTrader 8 section. When running into unexpected behavior please create a bug report in the NinjaTrader support forum's bug report section. Bug reports will be reviewed by our support and development staff inside of a 48 hour period. If you are unable to post to the bug section of the forum please contact [platformsupport@ninjatrader.com](mailto:platformsupport@ninjatrader.com).

tog_minus        [Installation Guide](javascript:HMToggle('toggle','Beta','Beta_ICON'))

|  |  |
| --- | --- |
| **Minimum PC Requirements**  You must meet the below minimum PC requirements to run NinjaTrader 8 Beta.    •Windows Vista (SP2) w/ Platform Update, Windows 7, Windows 8, Windows 10, Windows Server 2008 w/ Platform Update, Windows Server 2008 R2 or later.  •1 (GHz) or faster 32-bit or 64-bit processor  •2 GB RAM  •Microsoft .NET Framework 4.5  •Screen resolution of 1024 x 768 (16-bit color depth)    **Recommended PC Requirements**  NinjaTrader 8 was designed to take full advantage of modern PC architecture. To achieve the highest possible level of performance, NinjaTrader 8 will utilize all available CPU cores and additional memory resources. Depending on your actual usage with NinjaTrader you may need more or less resources as there is no one size fits all recommendation. Note: If you do plan on running strategy optimizations then having additional memory will be of direct benefit, the amount of additional memory needed is proportional to the number of CPU cores available.    •2 (GHz) or faster quad core 64-bit processor  •8 GB RAM  •DirectX10 compatible graphics card  •SSD Hard Drive    **Installation**  Please follow the below steps to get up and running on the beta.    1.If you do not have the Microsoft .NET Framework 4.5 installed you must download and install (Pre-installed on most PC’s and can be downloaded here: [Microsoft .NET Framework](http://www.microsoft.com/en-us/download/details.aspx?id=30653)).  2.[Download](http://ninjatrader.com/PlatformDirect) and install the NinjaTrader 8 Beta.  3.If you are running a 64-bit PC you will want to start the 'NinjaTrader 8 (64-bit)' version, on a 32-bit PC start the 'NinjaTrader 8 (32-bit)' version.  4.Firewall Software – NinjaTrader contacts our license server on application start up for license key validation. If you have a firewall, spyware or other such software running on your PC, please ensure that you grant NinjaTrader permission to access the internet or you may receive an invalid or license expired message.  5.Platform Activation for Live Trading – If you registered to use the free simulation version, or do not plan on trading live with NinjaTrader 8 Beta, you can skip this step.  oA license key to activate live trading is sent via email within minutes from the time you complete your purchase (check your junk/spam folder if you have not received it)  oStart NinjaTrader, select the menu Help > License Key within the Control Center and enter your license key  6.If you are a user of NijaTrader 7 you have the option to migrate NinjaTrader 7 data. The migration process will import historical data, replay data, instruments, trade history, account connections, market analyzer templates and strategy wizard templates. Any workspaces, chart templates, NinjaScript will unfortunately not be migrated over.  7.If you choose to migrate your connection should be immediately available in the Control Center -> Connections menu. If you did not migrate and need to setup your first connection please review the [connect to your account](https://ninjatrader.com/es/support/helpGuides/nt8/connecting_to_youraccount.htm) page to get started.     |  | | --- | | **Note:**Simulation color by default is transparent in NinjaTrader 8. If you migrate your settings from NinjaTrader 7 then simulation color will be maintained as is from NinjaTrader 7. You can change simulation color via tools > Option > Trading Category > Simulation Color. | |

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

tog_minus        [Expectations during beta](javascript:HMToggle('toggle','Betaexpectation','Betaexpectation_ICON'))

|  |  |
| --- | --- |
| **Help guide status**  We have made significant progress on the Help Guide and will continue to work on it throughout the beta process but it is still “Work In Progress”.  Any section marked at the top of the page with "Under Construction" is still being worked on and my contain errors; sections marked as "Needs Review" should be considered complete, but may change in the future.  Please feel free to report feedback to [platformsupport@ninjatrader.com](mailto:platformsupport@ninjatrader.com)    **Historical data status**  If you are connecting to CQG, Continuum or the FXCM then you are using historical data provided by NinjaTrader. Our Historical Data Servers for NinjaTrader 8 have been redesigned and are also in a beta testing period. As such, it is possible to experience outages and gaps in data.    **Ongoing Performance Optimization**  We are not 100% complete in our User Interface performance optimizations and as such, you may notice on large workspaces and/or slower hardware areas that will be improved.    **Live Trading with NinjaTrader 8 Beta**  There are risks associated with live trading on any [electronic trading platform](https://ninjatrader.com/es/support/helpGuides/nt8/risks_of_electronic_trading_wi.htm). While the NinjaTrader 8 live trading components are feature complete, there could be some undiscovered bugs which could adversely impact your ability to trade live. As such, the risks associated with live trading should be heavily considered.  We do have users that are reporting positive experiences with live trading and we have been using NinjaTrader 8 in a production capacity with CQG/Continuum on our own order desk for the past eight months. We continue to work through reported issues in anticipation of giving our users the best possible experience when NinjaTrader 8 is released into production.    **Reporting Bugs**  When running into unexpected behavior please note the following if possible and include it in your bug report as it will increase our ability to diagnose and resolve the issue.    •The time and date the bug occurred.  •Step by step actions (if any) that you took prior to the bug.  •Screen shot or video of the bug.     |  | | --- | | **Tip**:  Internally we use a free service called ['Jing'](https://www.techsmith.com/jing.html) which is very helpful to allow you to take a screen shot or record a video and share it. |     **How long will the beta last?**  Beta is a fluid process where we will continue to stay in beta for as long as we are receiving critical bug reports. We will be sure to stay in contact with you to provide you with new builds and keep you updated as to the progress of the beta. |

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

tog_minus        [NinjaScript](javascript:HMToggle('toggle','BetaHistoricalData','BetaHistoricalData_ICON'))

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **What's New with NinjaScript in NinjaTrader 8**  One of the primary goals of NinjaTrader 8 is to expand the number of NinjaScript objects, methods, and properties that are documented and officially supported, in order to provide our customers more control and flexibility than ever before. To that end, we have increased support across the board and unlocked additional NinjaScript types.     |  |  | | --- | --- | | **NinjaScript Type** | **Description** | | [Add Ons](https://ninjatrader.com/es/support/helpGuides/nt8/add_on.htm) | Add Ons run as a window is created. Add Ons can be used to extend the functionality of an existing NinjaTrader window, or to create your own window that enjoys all the features of others, such as workspace persistence and styling. | | [Bars Type](https://ninjatrader.com/es/support/helpGuides/nt8/bars_type.htm) | Bar Types are now fully supported in NinjaTrader 8. Although we allowed them to be customized in NinjaTrader 7, we offered no official support and provided only a small number of available slots. We have addressed both of these items with NinjaTrader 8. | | [Chart Style](https://ninjatrader.com/es/support/helpGuides/nt8/chart_style.htm) | Chart Styles are now fully supported and customizable. | | [Drawing Tools](https://ninjatrader.com/es/support/helpGuides/nt8/drawing_tools.htm) | You can now create your own custom drawing tools for use on NinjaTrader charts. | | [Import Type](https://ninjatrader.com/es/support/helpGuides/nt8/import_type.htm) | Allows you to customize the Historical Data Importer to allow different data formats to be imported in NinjaTrader 8. | | [Indicator](https://ninjatrader.com/es/support/helpGuides/nt8/indicator.htm) | Indicators were fully supported in NinjaTrader 7, and continue to be fully supported with increased functionality in NinjaTrader 8. | | [Market Analyzer Column](https://ninjatrader.com/es/support/helpGuides/nt8/market_analyzer_column.htm) | Market Analyzer Columns were allowed to be customized in NinjaTrader 7, but there was no official support or documentation. We have added these as fully supported in NinjaTrader 8. | | [Optimization Fitness](https://ninjatrader.com/es/support/helpGuides/nt8/optimization_fitness.htm) | An Optimization Fitness allows you to expand the optimizer by defining new ways to find the best strategy performance. | | [Optimizer](https://ninjatrader.com/es/support/helpGuides/nt8/optimizer.htm) | We support the standard (Brute Force) optimizer and a Genetic Optimizer, and we've opened up full support for writing your own optimizer. | | [Performance Metrics](https://ninjatrader.com/es/support/helpGuides/nt8/performance_metrics.htm) | A new NinjaScript component which allows you to expand performance metrics and define your own custom metrics. These will be viewable in the Trade Performance window and the Strategy Analyzer. | | [Share Service](https://ninjatrader.com/es/support/helpGuides/nt8/share_service.htm) | A new NinjaScript component to allow the creation of additional sharing service adapters. By default we support email, Facebook, Twitter, and Stocktwits. | | [Strategy](https://ninjatrader.com/es/support/helpGuides/nt8/strategy.htm) | Strategies were fully supported in NinjaTrader 7, and continue to be fully supported with increased functionality in NinjaTrader 8. | | [SuperDOM Column](https://ninjatrader.com/es/support/helpGuides/nt8/superdom_column.htm) | New NinjaScript component that allows you to create your own custom column to be added to a SuperDOM window. |       **Converting NinjaScript From NinjaTrader 7 to NinjaTrader 8**  In line with the goals of providing expanded support, consistency, performance, and improved functionality, we found it necessary to make code breaking changes. This means that existing scripts for NinjaTrader 7 will have to go through a porting process to get the code compiled and working in NinjaTrader 8. The changes required are typically mild and primarily constitute renaming methods and moving code to the new "State" system. We have compiled an overview guide of the new concepts in NinjaTrader and a list of the [code breaking changes](https://ninjatrader.com/es/support/helpGuides/nt8/code_breaking_changes.htm) to help you get started. We are very welcome to feedback to help improve these documents. If you run into issues porting a script, please post in our beta forums so that we can assist and improve documentation where needed.    **DLL Protection with CLI Secure**  For those who distribute NinjaScript assemblies, we upgraded our protection to use the latest Agile.net 6.3 protection. As the protection is stronger, the service is no longer free, and we have negotiated a discounted price of $100 for protecting NinjaScript assemblies. Please visit the [Protection / DLL Security](https://ninjatrader.com/es/support/helpGuides/nt8/protection_dll_security.htm) section of the help guide for more information. |

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

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

The **Condition Builder** is a very powerful feature that allows you to define complex conditions for your automated trading systems without having to know how to program.

tog_minus        [Understanding the Condition Builder](javascript:HMToggle('toggle','UnderstandingTheConditionBuilder','UnderstandingTheConditionBuilder_ICON'))

|  |
| --- |
| **Condition Builder**  Most if not all automated trading system code wizards are limited in scope in that they provide canned predefined expressions and only allow you to change a few parameters on those expressions. The NinjaTrader **Condition Builder** is advanced in that you can develop powerful expressions without limitations. Due to its power and flexibility, it is extremely important that you read through and understand its capabilities.    The **Condition Builder**is also a very powerful aid for those of you learning NinjaScript or learning how to program. You can build your conditions within the **Condition Builder** and instantly see NinjaScript code generated by having the [NinjaScript Editor](https://ninjatrader.com/es/support/helpGuides/nt8/editor.htm) open (by pressing the **View Code...** button in the Builder screen).    The **Condition Builder** can be accessed via the [Conditions and Actions](https://ninjatrader.com/es/support/helpGuides/nt8/builder_screens.htm) screen in the NinjaTrader Strategy Builder.    **Basic Operation**  The general concept of the **Condition Builder** is to generate a Boolean expression also known as comparison expressions or conditional expressions. What does that mean? It is simply an expression that results in a value of either TRUE or FALSE. For example, the expression    **2 < 7 (2 is less than 7)**    is a Boolean expression because the result is TRUE. All expressions that contain relational operators are Boolean. Boolean expressions or "Conditions" as they are known in NinjaTrader is used to determine when to take a specified action such as submitting an order or drawing on the chart.    Looking at the image below, you can instantly see that the **Condition Builder** is set up like a Boolean expression. Select an item from the left window (1), compare it to a selected item in the right window (1) and then select the relational operator (2).    Strategy_Builder_CB1    1. Available items such as indicators, price data, etc. to use for the comparison  2. List of relational operators (the relational operator will *only list the comparisons applicable* to your comparison item choices *and only after both sides* have been selected) |

[permalink](https://ninjatrader.com/es/support/helpGuides/nt8/index.html?condition_builder2.htm#UnderstandingTheConditionBuilder)

tog_minus        [How to make price data comparisons](javascript:HMToggle('toggle','HowToMakePriceDataComparisons','HowToMakePriceDataComparisons_ICON'))

|  |
| --- |
| **Price Data Comparisons**  You can compare a bar's price data such as checking for a higher close. The following is an example and represents one of many possible combinations.    1. Expand the **Price** category on the left side and select the **Close**.  2. Expand the **Price** category on the right side and select the **Close**.  3. Select the **greater** relational operator  4. Set the **Bars ago** parameter to a value of "1"    Strategy_Builder_CB2  Once the **OK** button is pressed, a condition is created that would translate to the following:    **"Current closing price is greater than the closing price of 1 bar ago"** |

[permalink](https://ninjatrader.com/es/support/helpGuides/nt8/index.html?condition_builder2.htm#HowToMakePriceDataComparisons)

tog_minus        [How to offset an item value](javascript:HMToggle('toggle','HowToOffsetAnItemValue','HowToOffsetAnItemValue_ICON'))

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Offsetting an Item Value**  You can offset the value of most items available in the **Condition Builder**. An offset is a value that is added, subtracted, multiplied or divided from / into the actual item's value. When an item is selected such as an indicator or price data, the **Offset** and **Offset** **type**parameters become visible in the window directly below the item selected. This is shown as numbers 5 and 6 in the image below.    **Offset type** can be set to:     |  |  | | --- | --- | | Arithmetic | Offsets by an arithmetic equation you can setup by the absolute value and the arithmetic offset operator to the left (+ - \* /) | | Pips | Offsets by the specified amount of pips | | Percent | Offsets a percentage value of the item's value. A value of 1 is equal to 100% where a value of 0.1 is equal to 10%. | | Ticks | Offsets by the specified amount of ticks |     Once the **Offset type** is selected, you must set the value **Offset**. In addition to the example below, you can see the "*Checking for Volume Expansion*" section below for another example that uses the **Percent** **Offset type**.    The following is an example and represents one of many possible combinations:    1. Expand the **Price** category and select the **Close**  2. Expand the **Price** category and select the **High**  3. Select the **greater** relational operator  4. Set the **Bars ago** parameter to a value of "1"  5. Set the **Offset type** parameter to **Ticks**  6. Set the **Offset** parameter to a value of "1"    Strategy_Builder_CB3    Once the **OK** button is pressed, a condition is created that would translate to the following:    **"Current closing price is greater than the high price of 1 bar ago + 1 tick"** |

[permalink](https://ninjatrader.com/es/support/helpGuides/nt8/index.html?condition_builder2.htm#HowToOffsetAnItemValue)

tog_minus        [How to make indicator to value comparisons](javascript:HMToggle('toggle','HowToMakeIndicatorToValueComparisons','HowToMakeIndicatorToValueComparisons_ICON'))

|  |
| --- |
| **Indicator to Value comparisons**  You can compare an indicator's value to a numeric value. This can come in handy if you wanted to check if ADX is over a value of 30 (trending) or if Stochastics is under a value of 20 (oversold) or any other conditions you can think of.    The following is an example and represents one of many possible combinations:    1. Expand the **Indicator** category and select the **ADX** indicator  2. Set the parameters of the indicator, for our example with the default values no changes are needed  3. Expand the **Misc**category and select **Numeric value**  4. Select the **greater** relational operator  5. Enter the numeric value you want to compare the indicator to (30 in our example)    Strategy_Builder_CB4  Once the **OK** button is pressed, a condition is created that would translate to the following:    **"Current value of a 14 period ADX is greater than 30"** |

[permalink](https://ninjatrader.com/es/support/helpGuides/nt8/index.html?condition_builder2.htm#HowToMakeIndicatorToValueComparisons)

tog_minus        [How to compare plot values of multi-plot indicators](javascript:HMToggle('toggle','HowToComparePlotValuesOfMultiplotIndicators','HowToComparePlotValuesOfMultiplotIndicators_ICON'))

|  |
| --- |
| **Comparing Plot Values of Multi-Plot indicators**  You can compare plots in the same indicator or select any individual plot within an indicator to create a condition.    The following is an example and represents one of many possible combinations:    1. Expand the **Indicator** category and select the **Stochastics** indicator  2. Set the indicator input parameters and select the **K** plot (green arrow)  3. Expand the **Indicator** category and select the **Stochastics** indicator  4. Select the **greater** relational operator  5. Set the indicator input parameters and select the **D** plot (green arrow)    Strategy_Builder_CB5  Once the **OK** button is pressed, a condition is created that would translate to the following:    **"Current K plot value of a Stochastics indicator is greater than the current D plot value of the same Stochastics indicator"** |

[permalink](https://ninjatrader.com/es/support/helpGuides/nt8/index.html?condition_builder2.htm#HowToComparePlotValuesOfMultiplotIndicators)

tog_minus        [How to use user inputs & variables](javascript:HMToggle('toggle','HowToUseUserDefinedInputsAndVariables','HowToUseUserDefinedInputsAndVariables_ICON'))

|  |
| --- |
| **User Inputs & Variables**  User inputs are simply variables that can be used in place of absolute values. They increase the flexibility of your strategy since you can substitute a variable for the period parameter of a simple moving average instead of provide an absolute value.    SMA(9) is how you express a 9 period simple moving average in NinjaScript. If you run a strategy, you would always be using a 9 period simple moving average. At run time, you might want to change this value to 10. User defined inputs accomplish this. If you created an input named "MyInput", you could express the simple moving average as SMA(MyInput). At run time, you can then configure your strategy by setting the value of "MyInput" to whatever value you like. In addition, user inputs are required when [optimizing a strategy](https://ninjatrader.com/es/support/helpGuides/nt8/optimize_a_strategy.htm).    User variables (not to be confused with inputs) behave in the same manner with the exception that they can not be configured when starting a strategy but can only be set programmatically during run time.    •User inputs are created from the [Builder screen](https://ninjatrader.com/es/support/helpGuides/nt8/builder_screens.htm)  •User variables can be set in the strategy logic through the **Condition Builder** (see the sections above)    The following is an example and represents one of many possible combinations, the example demonstrates the use of a user input however the sample approach applies to user variables.    1. Expand the **Price** category and select the **Close**.  2. Expand the **Indicator** category and select the **SMA** indicator  3. Select the **greater** relational operator  4. Set the **Period** parameter to a user defined input by pressing the "**Set**" button (green arrow) to open the **Value** window    Strategy_Builder_CB6  5. Expand the **User input** category and select the value **MAPeriod** and press the **OK** button    Strategy_Builder_CB7    6. The Condition Builder will now look as per the image below with the user input "MAPeriod" assigned to the parameter Period. When you apply this strategy to a chart, you will be able to set the value for the user input directly from the UI which will then be used to drive the SMA indicator.    Strategy_Builder_CB8    Once the **OK** button is pressed, a condition is created that would translate to the following:    **"Current closing price is greater than the user defined Period simple moving average"** |

[permalink](https://ninjatrader.com/es/support/helpGuides/nt8/index.html?condition_builder2.htm#HowToUseUserDefinedInputsAndVariables)

tog_minus        [How to create a cross over condition](javascript:HMToggle('toggle','HowToCreateACrossOverCondition','HowToCreateACrossOverCondition_ICON'))

|  |
| --- |
| **Cross Over Conditions**  You can check for either a **CrossAbove** or **CrossBelow** condition with a user defined look back period. The look back period sets the number of bars to look back to check for the cross over condition.    The following is an example and represents one of many possible combinations.    1. Expand the **Indicator** category and select the **EMA** indicator  2. Set the **Period** parameter to the desired value ("9" is used in this example)  3. Expand the **Indicator** category and select the **EMA** indicator  4. Set the **Period** parameter to the desired value ("20" is used in this example)  5. Select **CrossAbove** relational operator  6. Set the **Look back period**    Strategy_Builder_CB9  Once the **OK** button is pressed, a condition is created that would translate to the following:    **"9 period exponential moving average crosses above the 20 period exponential moving average in the last bar"** |

[permalink](https://ninjatrader.com/es/support/helpGuides/nt8/index.html?condition_builder2.htm#HowToCreateACrossOverCondition)

tog_minus        [How to use indicator inputs in other indicators](javascript:HMToggle('toggle','HowToUseIndicatorInputsInOtherIndicators','HowToUseIndicatorInputsInOtherIndicators_ICON'))

|  |
| --- |
| **Indicator on Indicator**  You can use indicators as input for other indicators ... actually, you can nest indicators within indicators infinitely if you really wanted to!    The following example is an example of applying a simple moving average (**SMA**) to a 14 period **ADX** indicator and is one of many possible combinations.    1. Expand the **Indicator** category and select the **SMA** indicator  2. Set Input series to the **ADX** indicator by pressing the "**Edit Input**" button to open the **Value** window  3. Select the **ADX** indicator and set any properties in the **Parameters** window    Strategy_Builder_CB10  3. Select the **ADX** indicator and set any properties in the **Properties** window  4. Press the **OK**button    Strategy_Builder_CB11  5. Once you have pressed the **OK** button, you will notice on the left lower window, the "Input series" parameters has now been set to the **ADX**(14) which is the 14 period **ADX** indicator.    Strategy_Builder_CB12 |

[permalink](https://ninjatrader.com/es/support/helpGuides/nt8/index.html?condition_builder2.htm#HowToUseIndicatorInputsInOtherIndicators)

tog_minus        [How to check for volume expansion](javascript:HMToggle('toggle','HowToCheckForVolumeExpansion','HowToCheckForVolumeExpansion_ICON'))

|  |
| --- |
| **Checking for Volume Expansion**  You can compare if the current bar's volume is greater than the prior bar's volume plus an offset amount.    The following is an example and represents one of many possible combinations.    1. Expand the **Indicator** category and select the **VOL** indicator  2. Expand the **Indicator** category and select the **VOL** indicator  3. Select the **greater than or equal**relational operator  4. Set the **Bars ago**parameter to a value of "1"  5. Set **Offset type** parameter to **Percent**  6. Set the **Offset** parameter to a value of "3" - *3 equals 300% percent here, i.e. 10% would be 0.1*    Strategy_Builder_CB13  Once the **OK** button is pressed, a condition is created that would translate to the following:    **"Current value of Volume is greater than or equal to the value of Volume of 1 bar ago + 300%"** |

[permalink](https://ninjatrader.com/es/support/helpGuides/nt8/index.html?condition_builder2.htm#HowToCheckForVolumeExpansion)

tog_minus        [How to create market position comparisons](javascript:HMToggle('toggle','HowToCreateMarketPositionComparisons','HowToCreateMarketPositionComparisons_ICON'))

|  |
| --- |
| **Creating Market Position Comparisons**  You can compare strategy state information such as but not limited to current market position or current position size.    The following is an example and represents one of many possible combinations.    1. Expand the **Strategy** category and select **Current market position**.  2. Expand the **Strategy** category and select **Market position**  3.Select the **equals to** relational operator  4. Select **Flat** from the Market position dropdown under Misc    Strategy_Builder_CB14    Once the **OK** button is pressed, a condition is created that would translate to the following:    **"Current market position equals flat"** |

[permalink](https://ninjatrader.com/es/support/helpGuides/nt8/index.html?condition_builder2.htm#HowToCreateMarketPositionComparisons)

tog_minus        [How to create time comparisons](javascript:HMToggle('toggle','HowToCreateTimeComparisons','HowToCreateTimeComparisons_ICON'))

|  |
| --- |
| **Creating Time Comparisons**  You can compare a bar's time data to a user defined time or date value.    The following is an example and represents one of many possible combinations.    Note: Time series represents a collection of bar Date / Time values of a bar series    1. Expand the **Time** category and select **Time series**  2. Expand the **Time** category and select **Time series**  3. Select the **greater than or equal**relational operator  4. Set the **Time** parameter to a user defined value of "10:00"    Strategy_Builder_CB15    Once the **OK** button is pressed, a condition is created that would translate to the following:    **"Current bar's time is greater or equal to 10:00 AM"** |

[permalink](https://ninjatrader.com/es/support/helpGuides/nt8/index.html?condition_builder2.htm#HowToCreateTimeComparisons)

tog_minus        [How to negate a condition](javascript:HMToggle('toggle','HowToNegateACondition','HowToNegateACondition_ICON'))

|  |
| --- |
| **Negating a Condition**  You can also negate a condition, so allowing for example to have a certain filter or technical indicator setup being the opposite and evaluate to false.    The following is an example and represents one of many possible combinations.    1. Expand the **Misc** category and select the **Cross above**  2. Click the **Series 1** input field and select the **DEMA** indicator as series for the cross comparison to use  3. Expand the **Misc** category and select the **False**  4. Select the equals relational operator    Strategy_Builder_CB16  Strategy_Builder_CB17    Once the **OK** button is pressed, a condition is created that would translate to the following:    **"The DEMA(14) indicator has not been crossed by the Close price within the last 10 bars"** |

[permalink](https://ninjatrader.com/es/support/helpGuides/nt8/index.html?condition_builder2.htm#HowToNegateACondition)

|  |  |
| --- | --- |
| **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#MiscellaneousPractices)

|  |  |
| --- | --- |
| **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) >  **Polarized Fractal Efficiency (PFE)** | [Previous page](https://ninjatrader.com/es/support/helpGuides/nt8/pivots.htm) [Return to chapter overview](https://ninjatrader.com/es/support/helpGuides/nt8/indicators.htm) [Next page](https://ninjatrader.com/es/support/helpGuides/nt8/price_oscillator.htm) |

**Description**

The Polarized Fractal Efficiency indicator uses fractal geometry to determine how efficiently the price is moving. When the PFE is zigzagging around zero, then the price is congested and not trending. When the PFE is smooth and above/below zero, then the price is in an up/down trend. The higher/lower the PFE value, the stronger the trend is.

... Courtesy of [FMLabs](http://www.fmlabs.com/reference/default.htm?url=PFE.htm)

**Syntax**

PFE(int *period, int smooth*)  
PFE(ISeries<double> *input*, int *period, int smooth*)

Returns default value  
PFE(int *period, int smooth*)[int *barsAgo*]  
PFE(ISeries<double> *input*, int *period, int smooth*)[int *barsAgo*]

**Return Value**

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

**Parameters**

|  |  |
| --- | --- |
| input | Indicator source data ([?](https://ninjatrader.com/es/support/helpGuides/nt8/valid_input_data_for_indicator.htm)) |
| period | Number of bars used in the calculation |
| smooth | The smoothing factor to be applied |

**Examples**

| ns |
| --- |
| // Prints the current value of a 20 period PFE using default price type double value = PFE(20, 2)[0]; Print("The current PFE value is " + value.ToString());   // Prints the current value of a 20 period PFE using high price type double value = PFE(High, 20, 2)[0]; Print("The current PFE value is " + value.ToString()); |

**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:**  [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) >  **Moving Average - Kaufman's Adaptive (KAMA)** | [Previous page](https://ninjatrader.com/es/support/helpGuides/nt8/moving_average_-_hull_hma.htm) [Return to chapter overview](https://ninjatrader.com/es/support/helpGuides/nt8/indicators.htm) [Next page](https://ninjatrader.com/es/support/helpGuides/nt8/moving_average_-_mesa_adaptive.htm) |

**Description**

Developed by Perry Kaufman, this indicator is an EMA using an Efficiency Ratio to modify the smoothing constant, which ranges from a minimum of Fast Length to a maximum of Slow Length.

**Syntax**

KAMA(int *fast*, int *period*, int *slow*)  
KAMA(ISeries<double> *input*, int *fast*, int *period*, int *slow*)

Returns default value  
KAMA(int *fast*, int *period*, int *slow*)[int *barsAgo*]  
KAMA(ISeries<double> *input*, int *fast*, int *period*, int *slow*)[int *barsAgo*]

**Return Value**

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

**Parameters**

|  |  |
| --- | --- |
| fast | Fast length |
| input | Indicator source data ([?](https://ninjatrader.com/es/support/helpGuides/nt8/valid_input_data_for_indicator.htm)) |
| period | Number of bars used in the calculation |
| slow | Slow length |

**Examples**

| ns |
| --- |
| // Prints the current value of a 20 period KAMA using default price type double value = KAMA(2, 20, 30)[0]; Print("The current KAMA value is " + value.ToString());   // Prints the current value of a 20 period KAMA using high price type double value = KAMA(High, 2, 20, 30)[0]; Print("The current KAMA value is " + value.ToString()); |

**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:**  [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

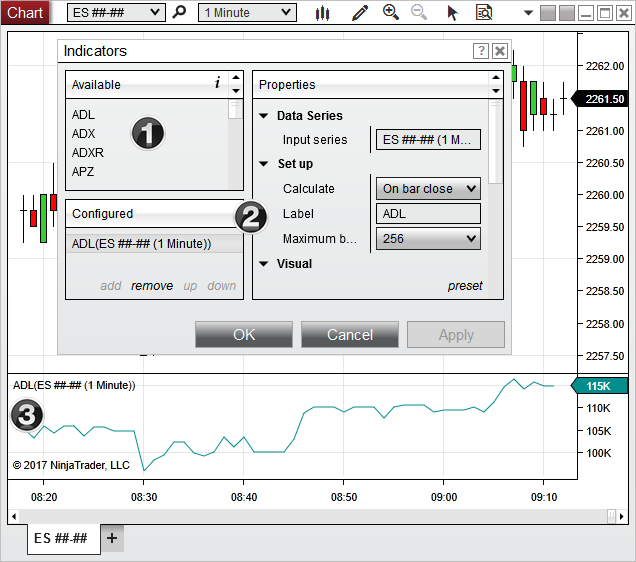
3.User configures desired **Properties** and presses "**Apply**" or "**OK**"

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



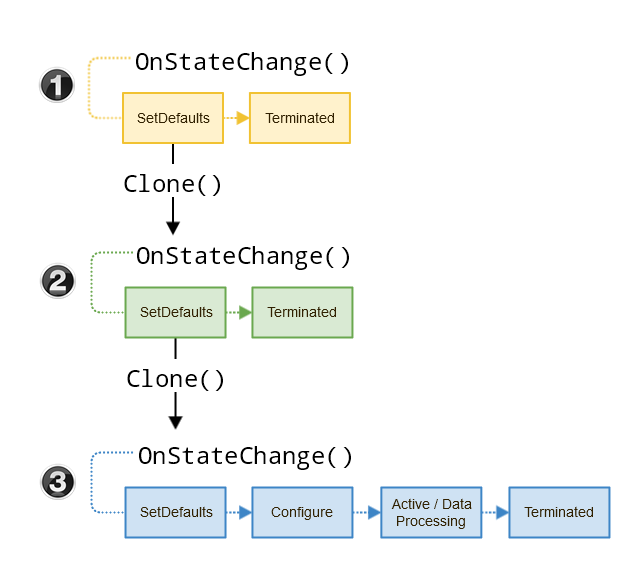
To visualize how each instance goes through its **States**, please consider the logic and flow chart below:

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) >  **OnStateChange()** | [Previous page](https://ninjatrader.com/es/support/helpGuides/nt8/marketdeptheventargs.htm) [Return to chapter overview](https://ninjatrader.com/es/support/helpGuides/nt8/common.htm) [Next page](https://ninjatrader.com/es/support/helpGuides/nt8/setstate.htm) |

**Definition**

An event driven method which is called whenever the script enters a new [State](https://ninjatrader.com/es/support/helpGuides/nt8/state.htm). The **OnStateChange()** method can be used to configure script properties, create one-time behavior when going from historical to real-time, as well as manage clean up resources on termination.

|  |
| --- |
| **Notes**:  •Viewing any UI element which lists NinjaScript classes (such as the Indicators or Strategies window, a chart's Chart Style dropdown menu, etc.) will initialize all classes of that Type when it is opened, which causes each script to enter **State.SetDefaults**, even if it is not actively configured or running in any window. It is important to keep this in mind when adding logic within **State.SetDefaults** in **OnStateChange()**, as this logic will be processed each time the script is initialized. For example, opening the Indicators window will trigger **State.SetDefaults** for all indicators in the system, and closing the Indicators window will trigger**State.Terminated** for all Indicators. In addition, disconnecting or connecting to a data provider can cause State transitions for any currently active scripts. Further discussion of this aspect of the state change model can be found via [*Understanding the lifecycle of your NinjaScript objects*](https://ninjatrader.com/es/support/helpGuides/nt8/understanding_the_lifecycle_of.htm).  •When an indicator is configured on a chart while a Compile is taking place in the NinjaScript Editor, it can appear that the script passes through **State.Terminated**. However, this is the result of a copy of the script being initialized at compile-time, NOT the result of the indicator on the chart being disabled and re-initialized. |

**Related Methods and Properties**

|  |  |
| --- | --- |
| [SetState()](https://ninjatrader.com/es/support/helpGuides/nt8/setstate.htm) | Method is used for changing the State of any running NinjaScript object. |
| [State](https://ninjatrader.com/es/support/helpGuides/nt8/state.htm) | Represents the current progression of the object as it advances from setup, processing data, to termination. |

**Method Return Value**

This method does not return a value.

**Syntax**  
See example below. The NinjaScript wizards automatically generate the method syntax for you.

Possible states are:

|  |  |  |
| --- | --- | --- |
| **State Name** | **This state is called when** | **This state is where you should** |
| State.SetDefaults | **SetDefaults** is always called when displaying objects in a UI list such as the Indicators dialogue window since temporary objects are created for the purpose of UI display | •Keep as lean as possible  •Set default values (pushed to UI) |
| State.Configure | **Configure** is called after a user adds an object to the applied list of objects and presses the OK or Apply button.  This state is called only once for the life of the object. | •Add additional data series via [AddDataSeries()](https://ninjatrader.com/es/support/helpGuides/nt8/adddataseries.htm)  •Declare custom resources |
| State.Active | **Active** is called once after the object is configured and is ready to process data (DrawingTools could see multiple calls as internally an object for hit testing is cloned) | •Used for objects such as [Share Service](https://ninjatrader.com/es/support/helpGuides/nt8/share_service.htm) which do not process price series data  •Indicate the object is ready to being processing information |
| State.DataLoaded | **DataLoaded** is called only once after all data series have been loaded. | •Use for logic that needs to access data related objects like Bars, Instruments, BarsPeriod, TradingHours or instantiating indicators  •Notify that all data series have been loaded  •Initialize any class level variables (including custom [Series<T>](https://ninjatrader.com/es/support/helpGuides/nt8/seriest.htm) objects) |
| State.Historical | **Historical**is called once the object begins to process historical data. This state is called once when running an object in real-time. This object is called multiple times when running a backtest optimization and the property [IsInstantiatedOnEachOptimizationIteration](https://ninjatrader.com/es/support/helpGuides/nt8/isinstantiatedoneachoptimizationiteration.htm) is true (default behavior) | •Notify that the object is processing historical data |
| State.Transition | **Transition** is called once as the object has finished processing historical data but before it starts to process realtime data. | •Notify that the indicator or strategy is is transitioning to realtime data  •Prepare realtime related resources |
| State.Realtime | **Realtime** is called once when the object begins to process realtime data. | •Notify that the indicator or strategy is processing realtime data  •Execute realtime related logic |
| State.Terminated | **Terminated** is called once when the object terminates. | •Notify the object is shutting down  •Use to clean up/dispose of resources |

**Active States vs Data Processing States**

After **State.Configure,**each type of NinjaScript type has its own state management system which can be classified under two categories:

•**Active state:**  State.Active

•**Data Processing states:**State.DataLoaded, State.Historical, State.Transition, State.Realtime

The table below lists each NinjaScript type and it's designed state management system:

|  |  |
| --- | --- |
| **NinjaScript Type** | **State Management System** |
| AddOns\* | Active state |
| BarTypes | Active state |
| ChartStyles | Active state |
| DrawingTools | Active state |
| Indicators | Data Processing states |
| ImportTypes | Active state |
| Market Analyzer Columns | Data Processing states |
| OptimizationFitnesses | Active state |
| Optimizers | Active state |
| PerformanceMetrics | Active state |
| ShareServices | Active state |
| Strategies | Data Processing states |
| SuperDOM Columns | Active state |

|  |
| --- |
| **Tips:**  •Resources created in **State.Configure** and not disposed of immediately will be kept and utilized if the NinjaScript object resides in grids (e.g. Strategy tab on Control Center), even if it is not enabled. Try to create resources in**State.Historical** or **State.DataLoaded** instead, if possible.  •**State.Historical** is called multiple times when running a backtest [optimization](https://ninjatrader.com/es/support/helpGuides/nt8/optimize_a_strategy.htm) on a strategy and the property "[IsInstantiatedOnEachOptimizationIteration](https://ninjatrader.com/es/support/helpGuides/nt8/isinstantiatedoneachoptimizationiteration.htm)" is **true** (default behavior).  • Scripts that require [Calculate](https://ninjatrader.com/es/support/helpGuides/nt8/calculate.htm) to be set by the developer must set this property in **State.Historical** in order to ensure that if this script is a child (hosted) that the parent.Calculate property which is adopted by the child is overridden again.  •When instantiating indicators in a [Multi-Series script](https://ninjatrader.com/es/support/helpGuides/nt8/multi-time_frame__instruments.htm) in OnStateChange, 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) |

**Examples**

| ns | |
| --- | --- |
| protected override void OnStateChange() {   if (State == State.SetDefaults)   {     // Calculate once at the end of every single bar     Calculate = Calculate.OnBarClose;         // Add two plots     AddPlot(Brushes.Blue, "Upper"));     AddPlot(Brushes.Orange, "Lower"));   }     else if (State == State.Configure)   {     // Adds a 5-minute Bars object to the strategy and is automatically assigned     // a Bars object index of 1 since the primary data the strategy is run against     // set by the UI takes the index of 0.             AddDataSeries("AAPL", BarsPeriodType.Minute, 5);       } } | |
| **Navigation:**  [NinjaScript](https://ninjatrader.com/es/support/helpGuides/nt8/ninjascript.htm) > [Distribution](https://ninjatrader.com/es/support/helpGuides/nt8/distribution.htm) > [Commercial Distribution](https://ninjatrader.com/es/support/helpGuides/nt8/commercial_distribution.htm) >  **Best Practices for Distribution** | | [Previous page](https://ninjatrader.com/es/support/helpGuides/nt8/licensing_user_authentication.htm) [Return to chapter overview](https://ninjatrader.com/es/support/helpGuides/nt8/commercial_distribution.htm) [Next page](https://ninjatrader.com/es/support/helpGuides/nt8/distribution_procedure.htm) |

The following are what we suggest for best practices for distribution.

**Do not deploy NinjaScript Source Files**

If you are a commercial vendor, you should never distribute the NinjaScript .cs source code files even if your IP is contained within an assembly or proprietary DLL. Source code files are editable by users and can result in unnecessary support issues.

**Naming Conventions**

Please use consistent naming convention with your indicators and strategies. We suggest adding a prefix to an indicator name. If your company name is "Hyper" you could name your indicators "HyperTrend" or "HyperOscillator" for example.

In the event that you provide NinjaScript export archives (zip files) as your means of distribution, NinjaTrader will automatically block incompatible scripts from importing so there will be no confusion by the user as to whether they are installing Version 7 or 8 scripts to their NinjaTrader installation. It is advisable to include the NinjaTrader version number in the export archive which will reduce potential support burden. For example, you could name your indicators “MyIndicator\_7.zip” and “MyIndicator\_8.zip”.

**Clean up your resources**

Always free up resources such as external windows DLL's or license management related resources. Resources should be freed within the [OnStateChange()](https://ninjatrader.com/es/support/helpGuides/nt8/onstatechange.htm) method in State.Terminate. NinjaTrader calls this method at the point at which a script is no longer used.

**User Authentication Trigger**

If you use a proprietary user authentication process, ensure that it is triggered within the [OnStateChange()](https://ninjatrader.com/es/support/helpGuides/nt8/onstatechange.htm) method in State.SetDefaults. This ensures that users are not forced to endure unnecessary delays on NinjaTrader start up or dialog windows that display available indicators and strategies as the windows are loaded. NinjaTrader, LLC provides a free licensing service for qualified 3rd party developers. For more information on this free service, contact your NinjaTrader Business Development representative.

**User Authentication Check State**

A license check should only be performed once and maintain its check state.

**User Authentication Time Out**

A license check should have a time out in case of internet issues, to enhance performance in this case.

**Custom Installer**

If you provide a custom installer, the installer should not overwrite any NinjaTrader deployed files, and you should provide an uninstall option which removes all installed files.

It is also preferred that you provide one installer that provides the user the option to install either a version 7 or version 8 compatible version of your product(s). Ensure that you only copy the correct files to the correct NinjaTrader installation folders since if you don’t it is possible that it could cause compile issues for the customer and it will be extremely difficult for all involved to isolate the cause.

These are the following folder names:

•Documents\NinjaTrader 7\bin\Custom

•Documents\NinjaTrader 8\bin\Custom

**Test on Legacy Operating Systems**

SomeNinjaTrader customers run on older Operating Systems (such as Windows 7) and you should make sure that your indicators, custom installers and external DLLs (if any are used) properly run on these legacy operating systems.

**Expose Indicator States**

If your proprietary indicator acts as a trend state (green bars are bullish, red bearish) its good practice to expose the indicators's state so that consumers of your indicators can use them within their own custom indicator or strategy.

|  |  |
| --- | --- |
| **Navigation:**  [NinjaScript](https://ninjatrader.com/es/support/helpGuides/nt8/ninjascript.htm) > [Distribution](https://ninjatrader.com/es/support/helpGuides/nt8/distribution.htm) > [Commercial Distribution](https://ninjatrader.com/es/support/helpGuides/nt8/commercial_distribution.htm) >  **Licensing/User Authentication** | [Previous page](https://ninjatrader.com/es/support/helpGuides/nt8/commercial_distribution.htm) [Return to chapter overview](https://ninjatrader.com/es/support/helpGuides/nt8/commercial_distribution.htm) [Next page](https://ninjatrader.com/es/support/helpGuides/nt8/best_practices.htm) |

NinjaTrader provides a free vendor license management service for user authentication to qualified 3rd party developers.

The service includes the following features:

•One method call within your NinjaScript indicator or strategy's constructor will enable the authentication process

•A NinjaScript AddOn dedicated to license management (Manage license, provide free trials)

•Licenses are exclusively tied to a combination of user-defined prefix + PC machine ID value, ensuring that licenses cannot be shared

•Manage all of your individual products, or group products together for licensing

•Licenses expire based on time/date

•Create free trial periods

For more information please contact [sales@ninjatrader.com](mailto:sales@ninjatrader.com) or your NinjaTrader Business Development representative. Once approved, you will receive a unique Vendor ID used to manage your user licenses, a Vendor Licensing Help Guide containing information, samples, and resources to guide you through the process of managing licensing.