NinjaScript Knowledge Base

Table of Contents

Part I: Introduction and Fundamentals

1. Introduction to NinjaScript
2. NinjaScript Basics and Fundamentals
3. NinjaScript Development Environment

Part II: Core Components

1. Indicators
2. Strategies
3. Drawing Tools
4. Candlestick Patterns

Part III: Trading Systems

1. Order Handling
2. ATM Strategies
3. Exit Strategies
4. Stop Types and Trailing Stops

Part IV: Advanced Topics

1. API Reference
2. Performance Optimization
3. Multi-Timeframe Analysis
4. Integration with External Data Sources
5. Machine Learning Integration

Part V: Conversion and Compatibility

1. Converting from PineScript to NinjaScript
2. Common Errors and Solutions

Part VI: Reference

1. Complete Indicator List and Settings
2. Common Design Patterns
3. Troubleshooting Guide

Introduction to NinjaScript

NinjaScript is a powerful programming language built on C# that allows traders to create custom technical indicators, automated trading strategies, drawing tools, and more within the NinjaTrader platform. This section provides an overview of NinjaScript and its capabilities.

What is NinjaScript?

NinjaScript is a robust programming language built on C# that allows you to create custom technical indicators, strategies, drawing tools, and more within the NinjaTrader platform. It offers a rich set of classes and methods specifically tailored for financial market analysis and trading system development.

NinjaScript provides a framework for developing custom trading applications within the NinjaTrader ecosystem. It leverages the power and flexibility of the C# programming language while providing specialized classes and methods for financial market analysis and trading.

Why Develop with NinjaScript?

By developing with NinjaScript, you can: - Customize your trading experience to fit your unique trading style and strategies - Automate trading strategies to eliminate emotional decision-making - Create custom indicators that aren't available in the standard package - Implement advanced risk management techniques - Backtest trading ideas against historical data - Integrate with external data sources and machine learning models - Create custom drawing tools for technical analysis - Develop custom columns for the SuperDOM and Market Analyzer - Build complete trading solutions with advanced order management

Prerequisites

To effectively develop with NinjaScript, you should have: - Basic understanding of programming concepts - Familiarity with C# syntax (or willingness to learn) - Understanding of trading concepts and technical analysis - NinjaTrader 8 platform installed - Visual Studio (recommended for advanced development) - .NET Framework knowledge (helpful but not required)

NinjaScript Components

NinjaScript allows you to develop various components for the NinjaTrader platform:

1. **Indicators**: Technical analysis tools that process price and volume data to generate signals or visualizations.
2. **Strategies**: Automated trading systems that can analyze market data and execute trades based on predefined rules.
3. **Drawing Tools**: Custom chart annotations and visualization tools.
4. **Chart Styles**: Custom representations of price data on charts.
5. **Bars Types**: Custom methods for aggregating and representing price data.

6. **Market Analyzer Columns**: Custom columns for the Market Analyzer window.
7. **SuperDOM Columns**: Custom columns for the SuperDOM order entry window.
8. **Import Types**: Custom data import formats for historical data.
9. **Optimization Fitness**: Custom metrics for strategy optimization.
10. **Performance Metrics**: Custom performance measurements for strategy analysis.
11. **Share Services**: Custom sharing capabilities for NinjaTrader content.
12. **Add-Ons**: Complete applications that extend NinjaTrader's functionality.

Getting Started with NinjaScript

To begin developing with NinjaScript:

1. **Install NinjaTrader**: Download and install the latest version of NinjaTrader 8.
2. **Explore the NinjaScript Editor**: Access it through NinjaTrader by selecting Tools > NinjaScript Editor.
3. **Study the Templates**: NinjaTrader provides templates for various NinjaScript components.
4. **Use the API Reference**: Familiarize yourself with the NinjaScript API through the Help > NinjaScript Reference menu.
5. **Start with Simple Projects**: Begin with basic indicators or strategies before attempting more complex projects.
6. **Join the Community**: Participate in the NinjaTrader forums to learn from other developers.

NinjaScript Development Best Practices

1. **Follow C# Conventions**: Adhere to standard C# naming and coding conventions.
2. **Optimize Performance**: Be mindful of performance implications, especially in real-time trading.
3. **Handle Errors Gracefully**: Implement proper error handling to prevent crashes.
4. **Comment Your Code**: Document your code thoroughly for future reference and sharing.
5. **Test Thoroughly**: Test your code with historical data before using it in live trading.
6. **Version Control**: Use version control systems to track changes to your code.
7. **Modular Design**: Create reusable components to simplify development and maintenance.
8. **Security Considerations**: Be cautious with external data sources and third-party libraries.

By understanding these fundamentals, you'll be well-prepared to begin your journey into NinjaScript development and create powerful trading tools tailored to your specific needs.

NinjaScript Basics and Fundamentals

This section covers the fundamental concepts and syntax of NinjaScript programming, providing a solid foundation for developing custom indicators and strategies.

NinjaScript Structure

NinjaScript follows an object-oriented programming model with a class-based structure. All

NinjaScript objects inherit from the `NinjaScriptBase` class and follow a specific lifecycle managed through state changes.

Class Hierarchy

The NinjaScript class hierarchy is organized as follows:

- `NinjaScriptBase`: The base class for all NinjaScript objects
- `AddOn`: For creating custom add-ons
- `BarsType`: For creating custom bars types
- `ChartStyle`: For creating custom chart styles
- `DrawingTool`: For creating custom drawing tools
- `ImportType`: For creating custom data import formats
- `Indicator`: For creating custom indicators
- `MarketAnalyzerColumn`: For creating custom Market Analyzer columns
- `OptimizationFitness`: For creating custom optimization fitness metrics
- `PerformanceMetric`: For creating custom performance metrics
- `ShareService`: For creating custom sharing services
- `Strategy`: For creating automated trading strategies
- `SuperDOMColumn`: For creating custom SuperDOM columns

Namespaces

NinjaScript code is organized into namespaces that reflect the component type:

```
namespace NinjaTrader.NinjaScript.Indicators
namespace NinjaTrader.NinjaScript.Strategies
namespace NinjaTrader.NinjaScript.DrawingTools
namespace NinjaTrader.NinjaScript.BarsTypes
namespace NinjaTrader.NinjaScript.ChartStyles
// And so on...
```

State Management

NinjaScript objects transition through various states during their lifecycle, managed through the `OnStateChange()` method:

1. **SetDefaults**: Initial state for setting default property values
2. Set default values for all properties
3. Define the name and description of your NinjaScript object
4. Configure display settings
5. **Configure**: Configuration state for adding indicators and setting up dependencies
6. Add plots for indicators
7. Initialize other indicators or data series

8. Set up any required resources
9. **Active**: Active state where the object is processing data
10. Object is now active but data may not be loaded yet
11. **DataLoaded**: State indicating that data has been loaded
12. Historical data is now available
13. Initialize any data-dependent resources
14. **Historical**: Processing historical data
15. Object is processing historical data
16. Perform historical calculations
17. **Transition**: Transitioning from historical to real-time
18. Transitioning between historical and real-time data
19. Prepare for real-time processing
20. **Realtime**: Processing real-time data
21. Object is processing real-time data
22. Handle real-time updates
23. **Terminated**: Object is being terminated
24. Clean up resources
25. Perform any necessary finalization

State Management Example

```
protected override void OnStateChange()
{
    if (State == State.SetDefaults)
    {
        Name = "My Custom Indicator";
        Description = "A simple custom indicator";
        Calculate = Calculate.OnBarClose;
        IsOverlay = false;

        // Set default property values
        Period = 14;
    }
    else if (State == State.Configure)
    {
        // Add plots
        AddPlot(Brushes.DodgerBlue, "MyPlot");
```

```
        // Add additional data series if needed
        AddDataSeries(BarsPeriodType.Minute, 5);
    }
    else if (State == State.DataLoaded)
    {
        // Initialize any data-dependent resources
    }
    else if (State == State.Terminated)
    {
        // Clean up resources
    }
}
```

Event Methods

NinjaScript provides several event methods that are called at specific times during execution:

OnBarUpdate()

The primary method for implementing calculation logic, called when a new bar is added or updated:

```
protected override void OnBarUpdate()
{
    // Skip calculation until we have enough bars
    if (CurrentBar < Period)
        return;

    // Calculate indicator value
    double value = SMA(Period)[0];

    // Assign value to plot
    Value[0] = value;
}
```

OnMarketData()

Called when market data is received:

```
protected override void OnMarketData(MarketDataEventArgs e)
{
    if (e.MarketDataType == MarketDataType.Last)
    {
        // Process last price
        double lastPrice = e.Price;
        long lastVolume = e.Volume;
```

```
        // Update values based on last price
    }
}
```

OnOrderUpdate()

Called when an order is updated (for strategies):

```
protected override void OnOrderUpdate(Order order, double limitPrice, double stc
    int quantity, int filled, double averageFillPrice,
    OrderState orderState, DateTime time, ErrorCode error, string comment)
{
    if (order.Name == "My Entry" && orderState == OrderState.Filled)
    {
        // Handle order fill
    }
}
```

OnExecutionUpdate()

Called when an execution occurs (for strategies):

```
protected override void OnExecutionUpdate(Execution execution, string executionI
    double price, int quantity, MarketPosition marketPosition,
    string orderId, DateTime time)
{
    // Handle execution
}
```

OnRender()

Called when rendering is needed (for custom drawing tools and chart styles):

```
protected override void OnRender(ChartControl chartControl, ChartScale chartScal
{
    // Custom rendering code
}
```

Properties and Methods

NinjaScript provides numerous properties and methods for accessing market data and performing calculations:

Data Access

```
// Price data
double closePrice = Close[0];  // Current close price
double highPrice = High[1];    // Previous high price
double lowPrice = Low[2];      // Low price from 2 bars ago
double openPrice = Open[3];    // Open price from 3 bars ago

// Volume data
double volume = Volume[0];     // Current volume

// Time data
DateTime time = Time[0];       // Current bar time

// Bar data
int barCount = CurrentBar;     // Number of bars processed
```

Indicator Methods

```
// Built-in indicators
double smaValue = SMA(14)[0];           // Simple Moving Average
double emaValue = EMA(14)[0];           // Exponential Moving Average
double rsiValue = RSI(14, 3)[0];        // Relative Strength Index
double stochValue = Stochastic(14, 3, 3).K[0]; // Stochastic

// Math functions
double maxValue = MAX(High, 10)[0];     // Maximum high over 10 bars
double minValue = MIN(Low, 10)[0];      // Minimum low over 10 bars
double sumValue = SUM(Close, 10)[0];    // Sum of closes over 10 bars
```

Strategy Methods

```
// Entry methods
EnterLong();                   // Enter long position
EnterShort();                  // Enter short position
EnterLong("My Entry", 2);      // Enter long with name and quantity

// Exit methods
ExitLong();                    // Exit long position
ExitShort();                   // Exit short position
ExitLong("My Exit", "My Entry"); // Exit specific entry

// Stop and target methods
SetStopLoss(CalculationMode.Ticks, 10); // Set stop loss
SetProfitTarget(CalculationMode.Ticks, 20); // Set profit target
```

Custom Properties

You can add custom properties to your NinjaScript objects using the `[NinjaScriptProperty]` attribute:

```
[NinjaScriptProperty]
[Range(1, 200)]
[Display(Name = "Period", Description = "The calculation period", Order = 1, Gro
public int Period { get; set; }

[NinjaScriptProperty]
[Display(Name = "Plot Style", Description = "The plot style", Order = 2, GroupNa
public PlotStyle MyPlotStyle { get; set; }

[NinjaScriptProperty]
[Display(Name = "Use Custom Color", Order = 3, GroupName = "Visual")]
public bool UseCustomColor { get; set; }

[XmlIgnore]
[Display(Name = "Plot Color", Description = "The color of the plot", Order = 4,
public Brush PlotColor { get; set; }

[Browsable(false)]
public string PlotColorSerializable
{
    get { return Serialize.BrushToString(PlotColor); }
    set { PlotColor = Serialize.StringToBrush(value); }
}
```

Best Practices for NinjaScript Development

1. **Follow the State Pattern**: Properly initialize and clean up resources in the appropriate states.
2. **Handle Edge Cases**: Always check for sufficient bars before performing calculations.
3. **Optimize Performance**: Minimize calculations in real-time processing.
4. **Use Descriptive Names**: Choose clear, descriptive names for your classes, methods, and properties.
5. **Document Your Code**: Add comments to explain complex logic.
6. **Test Thoroughly**: Test your code with different instruments and timeframes.
7. **Handle Errors Gracefully**: Implement proper error handling to prevent crashes.
8. **Avoid Hardcoding**: Use properties for values that might need to be changed.
9. **Follow C# Conventions**: Adhere to standard C# naming and coding conventions.
10. **Dispose Resources**: Properly dispose of any resources that implement IDisposable.

By understanding these fundamental concepts and following best practices, you'll be well-equipped to develop effective and efficient NinjaScript components.

NinjaScript Development Environment

This section covers the development environment for NinjaScript, including the NinjaScript Editor, debugging tools, and deployment options.

NinjaScript Editor Overview

The NinjaScript Editor is the integrated development environment (IDE) within NinjaTrader for creating and editing NinjaScript code. It provides a comprehensive set of tools for NinjaScript development.

Key Features

- **Syntax highlighting**: Colorizes code elements for better readability
- **IntelliSense code completion**: Suggests code completions as you type
- **Error highlighting**: Identifies syntax errors in real-time
- **Integrated debugging**: Tools for testing and troubleshooting code
- **Code snippets**: Pre-built code templates for common tasks
- **Project management**: Organize and manage multiple NinjaScript files
- **Auto-formatting**: Automatically formats code for consistency
- **Code navigation**: Easily navigate between methods and classes
- **Reference lookup**: Quick access to documentation and references

Editor Layout

The NinjaScript Editor is organized into several key areas:

1. **Solution Explorer**: Displays all NinjaScript projects and files
2. **Code Editor**: The main editing area for writing code
3. **Output Window**: Displays compilation results and debug output
4. **Error List**: Shows compilation errors and warnings
5. **Properties Window**: Displays and edits properties of selected items
6. **Toolbox**: Contains code snippets and templates

Editor Shortcuts

The NinjaScript Editor supports numerous keyboard shortcuts to enhance productivity:

- **F5**: Compile and run
- **F9**: Toggle breakpoint
- **Ctrl+Space**: Trigger IntelliSense
- **Ctrl+K, Ctrl+C**: Comment selected code
- **Ctrl+K, Ctrl+U**: Uncomment selected code
- **Ctrl+F**: Find
- **Ctrl+H**: Replace
- **F12**: Go to definition

- **Ctrl+.**: Show quick actions and refactorings

Debugging Tools

NinjaTrader provides several tools for debugging NinjaScript code:

Print Statements

The `Print()` method outputs values to the NinjaTrader Output window:

```
Print("Current value: " + Value[0]);
Print("SMA value: " + SMA(14)[0]);
```

Trace Levels

You can control the verbosity of debug output using trace levels:

```
// Only prints when trace level is set to high
if (TraceLevel == TraceLevel.High)
    Print("Detailed debug info: " + detailedInfo);
```

Visual Debugging

Use drawing objects to visualize values and conditions on charts:

```
// Draw an arrow at a signal point
if (signalCondition)
    Draw.ArrowUp(this, "Signal" + CurrentBar, false, 0, Low[0] - TickSize * 5, B
```

Breakpoints

Set breakpoints in the NinjaScript Editor to pause execution and inspect values:

1. Click in the margin next to the line where you want to set a breakpoint
2. When execution reaches the breakpoint, it will pause
3. Inspect variable values in the Locals window
4. Use Step Into, Step Over, and Step Out to control execution

Error Logging and Handling

Implement proper error handling to catch and log exceptions:

```
try
{
```

```
    // Code that might throw an exception
    double result = Calculate();
    Value[0] = result;
}
catch (Exception ex)
{
    Print("Error in calculation: " + ex.Message);
    // Handle the error gracefully
    Value[0] = double.NaN;
}
```

Deployment Options

Once you've developed your NinjaScript code, you can deploy it in several ways:

Local Compilation

The simplest deployment method is to compile and use within your local NinjaTrader installation:

1. In the NinjaScript Editor, press F5 or click the Compile button
2. If compilation succeeds, the component will be available in NinjaTrader
3. For indicators, add them to charts through the Indicators window
4. For strategies, access them through the Strategy Analyzer or Charts

Export as NinjaTrader Assembly

For distribution to other users, export as a NinjaTrader Assembly (.dll):

1. In NinjaTrader, select Tools > Import/Export NinjaScript
2. Select Export
3. Choose the components you want to export
4. Specify a file name and location
5. Click Export
6. The resulting .zip file can be distributed to other users

Protection and Licensing

To protect your intellectual property:

1. In NinjaTrader, select Tools > Import/Export NinjaScript
2. Select Export
3. Choose the components you want to export
4. Check "Apply protection" to obfuscate your code
5. Optionally, set a user-defined password
6. Click Export

Distribution Considerations

When distributing your NinjaScript components:

1. **Documentation**: Provide clear documentation on installation and usage
2. **Version Control**: Maintain version numbers and change logs
3. **Licensing**: Consider licensing terms for commercial distribution
4. **Support**: Establish a support channel for users
5. **Updates**: Plan for updates and bug fixes

NinjaTrader Ecosystem

For wider distribution, consider the NinjaTrader Ecosystem:

1. Create an account on the NinjaTrader Ecosystem website
2. Submit your component for review
3. Once approved, your component will be available to all NinjaTrader users
4. You can offer components for free or set a price

Development Best Practices

Code Organization

- Use meaningful namespaces to organize your code
- Group related components in the same project
- Use consistent naming conventions
- Separate business logic from UI code

Version Control

- Use a version control system like Git
- Commit changes regularly with descriptive messages
- Use branches for new features or major changes
- Consider hosting on GitHub or similar platforms

Testing

- Test with different instruments and timeframes
- Test with both historical and real-time data
- Verify calculations against known results
- Test edge cases and error conditions

Performance Optimization

- Minimize calculations in OnBarUpdate()
- Cache values that don't change frequently
- Use appropriate data structures
- Profile your code to identify bottlenecks

By leveraging these tools and following best practices, you can create robust, efficient, and reliable NinjaScript components for trading and analysis.

Indicators

This section covers the development of custom indicators in NinjaScript, including the indicator framework, common indicator types, and implementation examples.

Indicator Framework

The `Indicator` class extends `NinjaScriptBase` and provides functionality for creating custom indicators. Indicators are used to analyze price and volume data to identify potential trading opportunities or market conditions.

Key Properties and Methods

- **IsOverlay**: Determines if the indicator is displayed on the price chart or in a separate panel
- **Calculate**: Determines when indicator calculations are performed (on bar close, on price change, etc.)
- **OnBarUpdate()**: The primary method for implementing indicator logic
- **AddPlot()**: Method for adding plots to the indicator
- **Values[][]**: Multi-dimensional array for storing plot values
- **Value[]**: Shorthand for the first plot's values

Indicator Lifecycle

Indicators follow the standard NinjaScript lifecycle through the `OnStateChange()` method:

1. **SetDefaults**: Set default property values and configure basic settings
2. **Configure**: Add plots and initialize other indicators
3. **DataLoaded**: Initialize data-dependent resources
4. **Historical**: Process historical data
5. **Transition**: Transition from historical to real-time data
6. **Realtime**: Process real-time data
7. **Terminated**: Clean up resources

Basic Indicator Structure

```
using System;
using System.ComponentModel;
using System.Xml.Serialization;
using NinjaTrader.Cbi;
using NinjaTrader.Data;
using NinjaTrader.NinjaScript;

namespace NinjaTrader.NinjaScript.Indicators
```

```
{
    public class MyCustomIndicator : Indicator
    {
        protected override void OnStateChange()
        {
            if (State == State.SetDefaults)
            {
                Name = "My Custom Indicator";
                Description = "A simple custom indicator";
                Calculate = Calculate.OnBarClose;
                IsOverlay = false;

                // Set default property values
                Period = 14;
            }
            else if (State == State.Configure)
            {
                // Add plots
                AddPlot(Brushes.DodgerBlue, "MyPlot");
            }
        }

        protected override void OnBarUpdate()
        {
            // Skip calculation until we have enough bars
            if (CurrentBar < Period)
                return;

            // Calculate indicator value
            double value = SMA(Period)[0];

            // Assign value to plot
            Value[0] = value;
        }

        [NinjaScriptProperty]
        [Range(1, 200)]
        [Display(Name = "Period", Description = "The calculation period", Order
        public int Period { get; set; }
    }
}
```

Indicator Plots

Indicators can have multiple plots to display different values:

Adding Plots

```
// Add a simple plot with a color and name
AddPlot(Brushes.DodgerBlue, "MyPlot");

// Add a plot with more options
AddPlot(new Stroke(Brushes.Red, 2), PlotStyle.Line, "MySecondPlot");

// Add a histogram plot
AddPlot(new Stroke(Brushes.Green, 2), PlotStyle.Bar, "MyHistogram");
```

Plot Styles

NinjaTrader supports various plot styles:

- **Line**: A continuous line connecting values
- **Bar**: Vertical bars (histogram)
- **Cross**: Crosshairs at each value
- **Dot**: Dots at each value
- **Hash**: Hash marks at each value
- **HLine**: Horizontal lines at each value
- **Square**: Squares at each value
- **TriangleDown**: Downward-pointing triangles
- **TriangleUp**: Upward-pointing triangles

Accessing Plot Values

```
// Set the value of the first plot
Value[0] = calculatedValue;

// Set the value of a specific plot
Values[0][0] = firstPlotValue;
Values[1][0] = secondPlotValue;

// Access historical values
double previousValue = Value[1];  // Previous bar's value
double twoBarAgo = Value[2];      // Value from two bars ago
```

Common Indicator Types

NinjaTrader includes many built-in indicators that can be used as building blocks for custom indicators:

Moving Averages

```
// Simple Moving Average
double smaValue = SMA(14)[0];
```

```
// Exponential Moving Average
double emaValue = EMA(14)[0];

// Weighted Moving Average
double wmaValue = WMA(14)[0];

// Hull Moving Average
double hmaValue = HMA(14)[0];

// Triple Exponential Moving Average
double temaValue = TEMA(14)[0];
```

Oscillators

```
// Relative Strength Index
double rsiValue = RSI(14, 3)[0];

// Stochastic
double stochK = Stochastic(14, 3, 3).K[0];
double stochD = Stochastic(14, 3, 3).D[0];

// Moving Average Convergence Divergence
double macdValue = MACD(12, 26, 9).Diff[0];

// Commodity Channel Index
double cciValue = CCI(14)[0];

// Williams %R
double williamsR = WilliamsR(14)[0];
```

Volatility Indicators

```
// Bollinger Bands
double upperBand = Bollinger(2, 14).Upper[0];
double middleBand = Bollinger(2, 14).Middle[0];
double lowerBand = Bollinger(2, 14).Lower[0];

// Average True Range
double atrValue = ATR(14)[0];

// Keltner Channel
double keltnerUpper = Keltner(1.5, 14).Upper[0];
double keltnerMiddle = Keltner(1.5, 14).Middle[0];
double keltnerLower = Keltner(1.5, 14).Lower[0];
```

Volume Indicators

```
// Volume
double volumeValue = Volume[0];

// On Balance Volume
double obvValue = OBV()[0];

// Volume Moving Average
double volumeSMA = SMA(Volume, 14)[0];

// Chaikin Money Flow
double cmfValue = CMF(14)[0];
```

Trend Indicators

```
// Average Directional Index
double adxValue = ADX(14)[0];

// Parabolic SAR
double sarValue = ParabolicSAR(0.02, 0.2)[0];

// Ichimoku Cloud
double tenkanSen = Ichimoku(9, 26, 52).Tenkan[0];
double kijunSen = Ichimoku(9, 26, 52).Kijun[0];
double senkouSpanA = Ichimoku(9, 26, 52).SenkouSpanA[0];
double senkouSpanB = Ichimoku(9, 26, 52).SenkouSpanB[0];
```

Advanced Indicator Techniques

Multiple Data Series

Indicators can process multiple data series:

```
protected override void OnStateChange()
{
    if (State == State.SetDefaults)
    {
        // Standard setup
    }
    else if (State == State.Configure)
    {
        // Add an additional data series
        AddDataSeries(BarsPeriodType.Minute, 5);
    }
}
```

```
protected override void OnBarUpdate()
{
    // Check which data series is updating
    if (BarsInProgress == 0)
    {
        // Primary data series (e.g., 1-minute)
        // Process primary series data
    }
    else if (BarsInProgress == 1)
    {
        // Secondary data series (e.g., 5-minute)
        // Process secondary series data
    }
}
```

Custom Calculations

Implement custom calculations for specialized indicators:

```
private double CalculateCustomValue(int period)
{
    double sum = 0;
    double weight = 0;

    for (int i = 0; i < period; i++)
    {
        double value = Close[i];
        double currentWeight = period - i;

        sum += value * currentWeight;
        weight += currentWeight;
    }

    return weight > 0 ? sum / weight : 0;
}
```

Indicator Methods

Create reusable methods for complex calculations:

```
private double CalculateStandardDeviation(int period)
{
    double mean = SMA(period)[0];
    double sum = 0;

    for (int i = 0; i < period; i++)
    {
```

```
        sum += Math.Pow(Close[i] - mean, 2);
    }

    return Math.Sqrt(sum / period);
}
```

Implementation Examples

Simple Moving Average Crossover Indicator

```
public class SMACrossover : Indicator
{
    private SMA fastSMA;
    private SMA slowSMA;

    protected override void OnStateChange()
    {
        if (State == State.SetDefaults)
        {
            Name = "SMA Crossover";
            Description = "Detects crossovers between two SMAs";
            Calculate = Calculate.OnBarClose;
            IsOverlay = false;

            // Add parameters
            FastPeriod = 10;
            SlowPeriod = 20;
        }
        else if (State == State.Configure)
        {
            // Add plots
            AddPlot(Brushes.DodgerBlue, "FastSMA");
            AddPlot(Brushes.Red, "SlowSMA");
            AddPlot(Brushes.Green, "CrossoverSignal");

            // Initialize indicators
            fastSMA = SMA(FastPeriod);
            slowSMA = SMA(SlowPeriod);
        }
    }

    protected override void OnBarUpdate()
    {
        // Skip if not enough bars
        if (CurrentBar < SlowPeriod)
            return;

        // Calculate values
```

```
        double fastValue = fastSMA[0];
        double slowValue = slowSMA[0];

        // Detect crossover
        bool crossAbove = fastValue > slowValue && fastSMA[1] <= slowSMA[1];
        bool crossBelow = fastValue < slowValue && fastSMA[1] >= slowSMA[1];

        // Assign values to plots
        Values[0][0] = fastValue;
        Values[1][0] = slowValue;
        Values[2][0] = crossAbove ? 1 : (crossBelow ? -1 : 0);

        // Draw arrows for crossovers
        if (crossAbove)
            Draw.ArrowUp(this, "Up" + CurrentBar, false, 0, Low[0] - TickSize *
        else if (crossBelow)
            Draw.ArrowDown(this, "Down" + CurrentBar, false, 0, High[0] + TickSi
    }

    [NinjaScriptProperty]
    [Range(1, 200)]
    [Display(Name = "Fast Period", Description = "Period for the fast SMA", Orde
    public int FastPeriod { get; set; }

    [NinjaScriptProperty]
    [Range(1, 200)]
    [Display(Name = "Slow Period", Description = "Period for the slow SMA", Orde
    public int SlowPeriod { get; set; }
}
```

Relative Strength Comparison Indicator

```
public class RelativeStrengthComparison : Indicator
{
    private string symbolA;
    private string symbolB;

    protected override void OnStateChange()
    {
        if (State == State.SetDefaults)
        {
            Name = "Relative Strength Comparison";
            Description = "Compares the relative strength of two symbols";
            Calculate = Calculate.OnBarClose;
            IsOverlay = false;

            // Default parameters
            SymbolA = "MSFT";
```

```
            SymbolB = "AAPL";
            Period = 14;
        }
        else if (State == State.Configure)
        {
            // Add plots
            AddPlot(Brushes.DodgerBlue, "Ratio");
            AddPlot(Brushes.Red, "Signal");

            // Add data series for comparison symbol
            symbolA = Instrument.MasterInstrument.Name;
            symbolB = SymbolB;

            if (symbolA != symbolB)
                AddDataSeries(symbolB);
        }
    }

    protected override void OnBarUpdate()
    {
        // Wait until we have data for both symbols
        if (CurrentBars[0] < Period || CurrentBars[1] < Period)
            return;

        // Calculate relative strength ratio
        double priceA = Close[0];
        double priceB = Closes[1][0];

        if (priceB != 0)
        {
            double ratio = priceA / priceB;
            Values[0][0] = ratio;

            // Calculate signal line (EMA of ratio)
            Values[1][0] = EMA(Values[0], Period)[0];
        }
    }

    [NinjaScriptProperty]
    [Display(Name = "Symbol B", Description = "The comparison symbol", Order = 1
    public string SymbolB { get; set; }

    [NinjaScriptProperty]
    [Range(1, 200)]
    [Display(Name = "Period", Description = "The calculation period", Order = 2,
    public int Period { get; set; }
}
```

By understanding these concepts and examples, you can create powerful custom indicators to

enhance your trading analysis and decision-making.

Strategies

This section covers the development of automated trading strategies in NinjaScript, including the strategy framework, order management, and implementation examples.

Strategy Framework

The `Strategy` class extends `NinjaScriptBase` and provides functionality for creating automated trading strategies. Strategies can analyze market data and execute trades based on predefined rules.

Key Properties and Methods

- **OnBarUpdate()**: The primary method for implementing strategy logic
- **EnterLong()**, **EnterShort()**: Methods for entering positions
- **ExitLong()**, **ExitShort()**: Methods for exiting positions
- **SetStopLoss()**, **SetProfitTarget()**: Methods for setting exit conditions
- **OnOrderUpdate()**, **OnExecutionUpdate()**: Methods for handling order and execution events
- **PositionSize**: Property to set the default position size
- **DefaultQuantity**: Property to set the default order quantity
- **Calculate**: Determines when strategy calculations are performed

Strategy Lifecycle

Strategies follow the standard NinjaScript lifecycle through the `OnStateChange()` method:

1. **SetDefaults**: Set default property values and configure basic settings
2. **Configure**: Initialize indicators and other dependencies
3. **DataLoaded**: Initialize data-dependent resources
4. **Historical**: Process historical data for backtesting
5. **Transition**: Transition from historical to real-time data
6. **Realtime**: Process real-time data and execute live trades
7. **Terminated**: Clean up resources

Basic Strategy Structure

```
using System;
using System.ComponentModel;
using System.Xml.Serialization;
using NinjaTrader.Cbi;
using NinjaTrader.Data;
using NinjaTrader.NinjaScript;
```

```
namespace NinjaTrader.NinjaScript.Strategies
{
    public class MyCustomStrategy : Strategy
    {
        private SMA fastSMA;
        private SMA slowSMA;

        protected override void OnStateChange()
        {
            if (State == State.SetDefaults)
            {
                Name = "My Custom Strategy";
                Description = "A simple trading strategy";
                Calculate = Calculate.OnBarClose;

                // Set default property values
                FastPeriod = 10;
                SlowPeriod = 20;
                StopLossTicks = 20;
                ProfitTargetTicks = 40;
            }
            else if (State == State.Configure)
            {
                // Initialize indicators
                fastSMA = SMA(FastPeriod);
                slowSMA = SMA(SlowPeriod);
            }
        }

        protected override void OnBarUpdate()
        {
            // Skip if not enough bars
            if (CurrentBar < SlowPeriod)
                return;

            // Calculate values
            double fastValue = fastSMA[0];
            double slowValue = slowSMA[0];

            // Detect crossover
            bool crossAbove = fastValue > slowValue && fastSMA[1] <= slowSMA[1];
            bool crossBelow = fastValue < slowValue && fastSMA[1] >= slowSMA[1];

            // Entry logic
            if (crossAbove && Position.MarketPosition == MarketPosition.Flat)
            {
                EnterLong();
                SetStopLoss(CalculationMode.Ticks, StopLossTicks);
                SetProfitTarget(CalculationMode.Ticks, ProfitTargetTicks);
            }
            else if (crossBelow && Position.MarketPosition == MarketPosition.Fla
```

```
            {
                EnterShort();
                SetStopLoss(CalculationMode.Ticks, StopLossTicks);
                SetProfitTarget(CalculationMode.Ticks, ProfitTargetTicks);
            }
        }

        [NinjaScriptProperty]
        [Range(1, 200)]
        [Display(Name = "Fast Period", Description = "Period for the fast SMA",
        public int FastPeriod { get; set; }

        [NinjaScriptProperty]
        [Range(1, 200)]
        [Display(Name = "Slow Period", Description = "Period for the slow SMA",
        public int SlowPeriod { get; set; }

        [NinjaScriptProperty]
        [Range(1, 100)]
        [Display(Name = "Stop Loss (Ticks)", Description = "Stop loss distance i
        public int StopLossTicks { get; set; }

        [NinjaScriptProperty]
        [Range(1, 100)]
        [Display(Name = "Profit Target (Ticks)", Description = "Profit target di
        public int ProfitTargetTicks { get; set; }
    }
}
```

Order Management

NinjaScript provides a comprehensive order management system for strategies:

Entry Orders

```
// Simple market entry
EnterLong();                    // Enter long at market
EnterShort();                   // Enter short at market

// Named entries with quantity
EnterLong("My Long Entry", 2); // Enter long with name and quantity
EnterShort("My Short Entry", 2); // Enter short with name and quantity

// Limit and stop entries
EnterLongLimit("My Limit Entry", 2, limitPrice);
EnterShortLimit("My Limit Entry", 2, limitPrice);
EnterLongStop("My Stop Entry", 2, stopPrice);
EnterShortStop("My Stop Entry", 2, stopPrice);
```

```
// Stop-limit entries
EnterLongStopLimit("My StopLimit Entry", 2, stopPrice, limitPrice);
EnterShortStopLimit("My StopLimit Entry", 2, stopPrice, limitPrice);
```

Exit Orders

```
// Simple market exits
ExitLong();                    // Exit all long positions
ExitShort();                   // Exit all short positions

// Named exits
ExitLong("My Long Exit", "My Long Entry"); // Exit specific entry
ExitShort("My Short Exit", "My Short Entry"); // Exit specific entry

// Limit and stop exits
ExitLongLimit("My Limit Exit", "My Long Entry", limitPrice);
ExitShortLimit("My Limit Exit", "My Short Entry", limitPrice);
ExitLongStop("My Stop Exit", "My Long Entry", stopPrice);
ExitShortStop("My Stop Exit", "My Short Entry", stopPrice);

// Stop-limit exits
ExitLongStopLimit("My StopLimit Exit", "My Long Entry", stopPrice, limitPrice);
ExitShortStopLimit("My StopLimit Exit", "My Short Entry", stopPrice, limitPrice)
```

Stop Loss and Profit Target

```
// Set stop loss and profit target for all entries
SetStopLoss(CalculationMode.Ticks, 20);
SetProfitTarget(CalculationMode.Ticks, 40);

// Set stop loss and profit target for a specific entry
SetStopLoss("My Long Entry", CalculationMode.Ticks, 20);
SetProfitTarget("My Long Entry", CalculationMode.Ticks, 40);

// Different calculation modes
SetStopLoss(CalculationMode.Price, stopPrice);
SetProfitTarget(CalculationMode.Price, targetPrice);
SetStopLoss(CalculationMode.Percent, 1.0); // 1% stop loss
SetProfitTarget(CalculationMode.Percent, 2.0); // 2% profit target
```

Trailing Stops

```
// Set trailing stop loss
SetTrailStop(CalculationMode.Ticks, 20);
```

```
// Set trailing stop loss with step
SetTrailStop(CalculationMode.Ticks, 20, 5);

// Set trailing stop loss for a specific entry
SetTrailStop("My Long Entry", CalculationMode.Ticks, 20);
```

Order Handling Events

```
protected override void OnOrderUpdate(Order order, double limitPrice, double stc
    int quantity, int filled, double averageFillPrice,
    OrderState orderState, DateTime time, ErrorCode error, string comment)
{
    if (order.Name == "My Long Entry" && orderState == OrderState.Filled)
    {
        // Handle order fill
        Print("Order filled at " + averageFillPrice);
    }
    else if (orderState == OrderState.Rejected)
    {
        // Handle order rejection
        Print("Order rejected: " + error);
    }
}

protected override void OnExecutionUpdate(Execution execution, string executionI
    double price, int quantity, MarketPosition marketPosition,
    string orderId, DateTime time)
{
    // Handle execution
    Print("Execution at " + price + " for " + quantity + " contracts");
}
```

Position Management

Strategies can access and manage positions:

Position Properties

```
// Check current position
if (Position.MarketPosition == MarketPosition.Flat)
    Print("No position");
else if (Position.MarketPosition == MarketPosition.Long)
    Print("Long position of " + Position.Quantity + " contracts");
else if (Position.MarketPosition == MarketPosition.Short)
    Print("Short position of " + Position.Quantity + " contracts");
```

```
// Access position details
double avgPrice = Position.AveragePrice;
int quantity = Position.Quantity;
double profit = Position.GetUnrealizedProfitLoss(PerformanceUnit.Currency);
```

Multiple Positions

```
// For strategies with multiple positions
foreach (Position position in Account.Positions)
{
    if (position.MarketPosition != MarketPosition.Flat)
    {
        Print("Position in " + position.Instrument.FullName +
              ": " + position.MarketPosition +
              ", Quantity: " + position.Quantity);
    }
}
```

Strategy Optimization

NinjaTrader provides tools for optimizing strategy parameters:

Optimization Parameters

```
[NinjaScriptProperty]
[Range(5, 50, 5)]  // Min, Max, Step
[Display(Name = "Fast Period", Description = "Period for the fast SMA", Order =
public int FastPeriod { get; set; }

[NinjaScriptProperty]
[Range(10, 100, 10)]  // Min, Max, Step
[Display(Name = "Slow Period", Description = "Period for the slow SMA", Order =
public int SlowPeriod { get; set; }
```

Custom Optimization Fitness

```
// In a custom OptimizationFitness class
protected override void OnCalculatePerformanceValues()
{
    // Calculate custom fitness value
    double profitFactor = Performance.AllTrades.TradesPerformance.ProfitFactor;
    double netProfit = Performance.AllTrades.TradesPerformance.NetProfit;
    double drawdown = Performance.AllTrades.TradesPerformance.MaxDrawdown;
```

```
    // Custom fitness formula
    Value = (profitFactor * netProfit) / (drawdown > 0 ? drawdown : 1);
}
```

Implementation Examples

Moving Average Crossover Strategy

```
public class MACrossoverStrategy : Strategy
{
    private SMA fastSMA;
    private SMA slowSMA;

    protected override void OnStateChange()
    {
        if (State == State.SetDefaults)
        {
            Name = "MA Crossover Strategy";
            Description = "A strategy based on SMA crossovers";

            // Add parameters
            FastPeriod = 10;
            SlowPeriod = 20;
            StopLossTicks = 20;
            ProfitTargetTicks = 40;
        }
        else if (State == State.Configure)
        {
            // Initialize indicators
            fastSMA = SMA(FastPeriod);
            slowSMA = SMA(SlowPeriod);
        }
    }

    protected override void OnBarUpdate()
    {
        // Skip if not enough bars
        if (CurrentBar < SlowPeriod)
            return;

        // Calculate values
        double fastValue = fastSMA[0];
        double slowValue = slowSMA[0];

        // Detect crossover
        bool crossAbove = fastValue > slowValue && fastSMA[1] <= slowSMA[1];
        bool crossBelow = fastValue < slowValue && fastSMA[1] >= slowSMA[1];
```

```
        // Entry logic
        if (crossAbove && Position.MarketPosition == MarketPosition.Flat)
        {
            EnterLong();
            SetStopLoss(CalculationMode.Ticks, StopLossTicks);
            SetProfitTarget(CalculationMode.Ticks, ProfitTargetTicks);
        }
        else if (crossBelow && Position.MarketPosition == MarketPosition.Flat)
        {
            EnterShort();
            SetStopLoss(CalculationMode.Ticks, StopLossTicks);
            SetProfitTarget(CalculationMode.Ticks, ProfitTargetTicks);
        }
    }

    [NinjaScriptProperty]
    [Range(1, 200)]
    [Display(Name = "Fast Period", Description = "Period for the fast SMA", Orde
    public int FastPeriod { get; set; }

    [NinjaScriptProperty]
    [Range(1, 200)]
    [Display(Name = "Slow Period", Description = "Period for the slow SMA", Orde
    public int SlowPeriod { get; set; }

    [NinjaScriptProperty]
    [Range(1, 100)]
    [Display(Name = "Stop Loss (Ticks)", Description = "Stop loss distance in ti
    public int StopLossTicks { get; set; }

    [NinjaScriptProperty]
    [Range(1, 100)]
    [Display(Name = "Profit Target (Ticks)", Description = "Profit target distar
    public int ProfitTargetTicks { get; set; }
}
```

RSI Strategy with Multiple Timeframes

```
public class RSIMultiTimeframeStrategy : Strategy
{
    private RSI rsiShort;
    private RSI rsiLong;

    protected override void OnStateChange()
    {
        if (State == State.SetDefaults)
        {
```

```
            Name = "RSI Multi-Timeframe Strategy";
            Description = "A strategy based on RSI across multiple timeframes";

            // Add parameters
            ShortPeriod = 14;
            LongPeriod = 14;
            OverboughtLevel = 70;
            OversoldLevel = 30;
        }
        else if (State == State.Configure)
        {
            // Add a longer timeframe data series
            AddDataSeries(BarsPeriodType.Minute, 15);

            // Initialize indicators
            rsiShort = RSI(ShortPeriod, 3);
            rsiLong = RSI(LongPeriod, 3);
        }
    }

    protected override void OnBarUpdate()
    {
        // Process primary data series (e.g., 5-minute)
        if (BarsInProgress == 0)
        {
            // Skip if not enough bars or if the longer timeframe hasn't updated
            if (CurrentBars[0] < ShortPeriod || CurrentBars[1] < LongPeriod)
                return;

            // Get RSI values
            double rsiShortValue = rsiShort[0];
            double rsiLongValue = rsiLong[1]; // From the longer timeframe

            // Entry logic
            if (Position.MarketPosition == MarketPosition.Flat)
            {
                // Enter long when both RSIs are oversold
                if (rsiShortValue < OversoldLevel && rsiLongValue < OversoldLeve
                {
                    EnterLong();
                    SetStopLoss(CalculationMode.Percent, 1.0);
                    SetProfitTarget(CalculationMode.Percent, 2.0);
                }
                // Enter short when both RSIs are overbought
                else if (rsiShortValue > OverboughtLevel && rsiLongValue > Overb
                {
                    EnterShort();
                    SetStopLoss(CalculationMode.Percent, 1.0);
                    SetProfitTarget(CalculationMode.Percent, 2.0);
                }
            }
```

```
            // Exit logic
            else if (Position.MarketPosition == MarketPosition.Long)
            {
                // Exit long when short-term RSI becomes overbought
                if (rsiShortValue > OverboughtLevel)
                {
                    ExitLong();
                }
            }
            else if (Position.MarketPosition == MarketPosition.Short)
            {
                // Exit short when short-term RSI becomes oversold
                if (rsiShortValue < OversoldLevel)
                {
                    ExitShort();
                }
            }
        }
    }

    [NinjaScriptProperty]
    [Range(2, 50)]
    [Display(Name = "Short-Term RSI Period", Order = 1, GroupName = "Parameters"
    public int ShortPeriod { get; set; }

    [NinjaScriptProperty]
    [Range(2, 50)]
    [Display(Name = "Long-Term RSI Period", Order = 2, GroupName = "Parameters")
    public int LongPeriod { get; set; }

    [NinjaScriptProperty]
    [Range(50, 90)]
    [Display(Name = "Overbought Level", Order = 3, GroupName = "Parameters")]
    public int OverboughtLevel { get; set; }

    [NinjaScriptProperty]
    [Range(10, 50)]
    [Display(Name = "Oversold Level", Order = 4, GroupName = "Parameters")]
    public int OversoldLevel { get; set; }
}
```

By understanding these concepts and examples, you can create powerful automated trading strategies to execute your trading ideas with precision and consistency.

Drawing Tools

This section covers the development of custom drawing tools in NinjaScript, including the drawing framework, common drawing methods, and implementation examples.

Drawing Framework

NinjaTrader provides a rich set of drawing tools for visualizing data on charts. The `Draw` class includes methods for creating various drawing objects, and you can also develop custom drawing tools by extending the `DrawingTool` class.

Built-in Drawing Methods

The `Draw` class provides numerous methods for creating drawing objects on charts:

Lines and Rays

```
// Drawing a line between two points
Draw.Line(string tag, int startBarsAgo, double startY, int endBarsAgo, double er

// Drawing a ray (line extending infinitely in one direction)
Draw.Ray(string tag, int startBarsAgo, double startY, int endBarsAgo, double enc

// Drawing an extended line (line extending infinitely in both directions)
Draw.ExtendedLine(string tag, int startBarsAgo, double startY, int endBarsAgo, c
```

Shapes

```
// Drawing a rectangle
Draw.Rectangle(string tag, int startBarsAgo, double startY, int endBarsAgo, doub

// Drawing a triangle
Draw.Triangle(string tag, int barsAgo1, double y1, int barsAgo2, double y2, int

// Drawing an ellipse
Draw.Ellipse(string tag, int startBarsAgo, double startY, int endBarsAgo, double
```

Text and Annotations

```
// Drawing text
Draw.Text(string tag, string text, int barsAgo, double y, Brush brush);

// Drawing a text box
Draw.TextFixed(string tag, string text, TextPosition position, Brush brush);

// Drawing arrows
Draw.ArrowUp(string tag, bool isAutoScale, int barsAgo, double y, Brush brush);
Draw.ArrowDown(string tag, bool isAutoScale, int barsAgo, double y, Brush brush)
```

Fibonacci Tools

```
// Drawing Fibonacci retracements
Draw.FibonacciRetracements(string tag, int startBarsAgo, double startY, int endB

// Drawing Fibonacci extensions
Draw.FibonacciExtensions(string tag, int startBarsAgo, double startY, int middle

// Drawing Fibonacci time extensions
Draw.FibonacciTimeExtensions(string tag, int startBarsAgo, int endBarsAgo);
```

Advanced Tools

```
// Drawing Andrew's pitchfork
Draw.AndrewsPitchfork(string tag, int startBarsAgo, double startY, int middleBar

// Drawing Gann fan
Draw.GannFan(string tag, int barsAgo, double y);

// Drawing regression channel
Draw.RegressionChannel(string tag, int startBarsAgo, int endBarsAgo);
```

Managing Drawing Objects

```
// Removing a specific drawing object
RemoveDrawObject(string tag);

// Removing all drawing objects
RemoveDrawObjects();

// Removing drawing objects with a specific prefix
RemoveDrawObjects("Signal");

// Checking if a drawing object exists
bool exists = DrawObjects[tag] != null;

// Accessing a drawing object's properties
if (DrawObjects[tag] != null)
{
    DrawingTool drawingTool = DrawObjects[tag];
    // Access properties
}
```

Custom Drawing Tools

You can create custom drawing tools by extending the `DrawingTool` class:

Basic Structure

```
using System;
using System.ComponentModel;
using System.Windows;
using System.Windows.Input;
using System.Windows.Media;
using System.Xml.Serialization;
using NinjaTrader.Gui;
using NinjaTrader.Gui.Chart;
using NinjaTrader.Gui.Tools;

namespace NinjaTrader.NinjaScript.DrawingTools
{
    public class MyCustomDrawingTool : DrawingTool
    {
        // Define anchor points
        private ChartAnchor startAnchor;
        private ChartAnchor endAnchor;

        // Constructor
        public MyCustomDrawingTool()
        {
            startAnchor = new ChartAnchor();
            endAnchor = new ChartAnchor();
        }

        // Override methods for drawing and interaction
        public override void OnRender(ChartControl chartControl, ChartScale char
        {
            // Rendering code
        }

        // Other overrides for mouse interaction, etc.
    }
}
```

Implementing OnRender

The `OnRender` method is where you implement the drawing logic:

```
public override void OnRender(ChartControl chartControl, ChartScale chartScale)
{
    // Get device (screen) points from chart anchors
    Point startPoint = startAnchor.GetPoint(chartControl, chartScale);
    Point endPoint = endAnchor.GetPoint(chartControl, chartScale);
```

```
    // Create a pen for drawing
    SharpDX.Direct2D1.Brush strokeBrush = chartControl.ChartPanel.BarBrushes.Con
        chartControl.ChartPanel.BarBrushes[Stroke.BrushDX] :
        chartControl.ChartPanel.BarBrushes.Values.FirstOrDefault();

    SharpDX.Direct2D1.Brush areaBrush = chartControl.ChartPanel.BarBrushes.Conta
        chartControl.ChartPanel.BarBrushes[AreaBrush.BrushDX] :
        chartControl.ChartPanel.BarBrushes.Values.FirstOrDefault();

    // Draw a line
    RenderTarget.DrawLine(
        new SharpDX.Vector2((float)startPoint.X, (float)startPoint.Y),
        new SharpDX.Vector2((float)endPoint.X, (float)endPoint.Y),
        strokeBrush,
        (float)Stroke.Width,
        Stroke.StrokeStyle
    );

    // Draw other elements as needed
}
```

Mouse Interaction

Implement mouse interaction methods to make your drawing tool interactive:

```
public override void OnMouseDown(ChartControl chartControl, ChartPanel chartPane
{
    if (!IsVisible)
        return;

    // Handle mouse down event
    if (DrawingState == DrawingState.Building)
    {
        if (startAnchor.IsEditing)
        {
            startAnchor.IsEditing = false;
            endAnchor.IsEditing = true;

            // Set initial end anchor position
            endAnchor.CopyDataValues(dataPoint);
            endAnchor.CopyPointValues(dataPoint);
        }
        else if (endAnchor.IsEditing)
        {
            endAnchor.IsEditing = false;
            DrawingState = DrawingState.Normal;
        }
    }
```

```
}

public override void OnMouseMove(ChartControl chartControl, ChartPanel chartPane
{
    if (!IsVisible)
        return;

    // Handle mouse move event
    if (DrawingState == DrawingState.Building && endAnchor.IsEditing)
    {
        endAnchor.CopyDataValues(dataPoint);
        endAnchor.CopyPointValues(dataPoint);
    }
}
```

Properties

Define properties for your drawing tool:

```
[XmlIgnore]
[Display(Name = "Stroke", Order = 1, GroupName = "Visual")]
public Stroke Stroke { get; set; }

[Browsable(false)]
public string StrokeSerializable
{
    get { return Stroke.ToString(); }
    set { Stroke = new Stroke(value); }
}

[XmlIgnore]
[Display(Name = "Area Brush", Order = 2, GroupName = "Visual")]
public Brush AreaBrush { get; set; }

[Browsable(false)]
public string AreaBrushSerializable
{
    get { return Serialize.BrushToString(AreaBrush); }
    set { AreaBrush = Serialize.StringToBrush(value); }
}

[Display(Name = "Opacity", Order = 3, GroupName = "Visual")]
public int Opacity { get; set; }
```

Implementation Examples

Custom Support/Resistance Line Tool

```
public class SupportResistanceTool : DrawingTool
{
    private ChartAnchor startAnchor;
    private ChartAnchor endAnchor;

    public SupportResistanceTool()
    {
        startAnchor = new ChartAnchor();
        endAnchor = new ChartAnchor();

        // Set default properties
        Stroke = new Stroke(Brushes.DodgerBlue, 2);
        IsExtendedLine = true;
        LineStyle = LineStyle.Solid;
    }

    public override void OnRender(ChartControl chartControl, ChartScale chartSca
    {
        // Get device points
        Point startPoint = startAnchor.GetPoint(chartControl, chartScale);
        Point endPoint = endAnchor.GetPoint(chartControl, chartScale);

        // Get the brush
        SharpDX.Direct2D1.Brush strokeBrush = chartControl.ChartPanel.BarBrushes
            chartControl.ChartPanel.BarBrushes[Stroke.BrushDX] :
            chartControl.ChartPanel.BarBrushes.Values.FirstOrDefault();

        // Create stroke style
        SharpDX.Direct2D1.StrokeStyle strokeStyle = null;
        switch (LineStyle)
        {
            case LineStyle.Solid:
                strokeStyle = chartControl.ChartPanel.ChartControl.DashStyleDash
                break;
            case LineStyle.Dashed:
                strokeStyle = chartControl.ChartPanel.ChartControl.DashStyleDash
                break;
            case LineStyle.Dotted:
                strokeStyle = chartControl.ChartPanel.ChartControl.DashStyleDot;
                break;
        }

        // Draw the line
        if (IsExtendedLine)
        {
            // Calculate extended line points
            float slope = (float)(endPoint.Y - startPoint.Y) / (float)(endPoint.
            float yIntercept = (float)startPoint.Y - slope * (float)startPoint.X
```

```
            float leftX = 0;
            float leftY = yIntercept;
            float rightX = (float)chartControl.ActualWidth;
            float rightY = slope * rightX + yIntercept;

            // Draw extended line
            RenderTarget.DrawLine(
                new SharpDX.Vector2(leftX, leftY),
                new SharpDX.Vector2(rightX, rightY),
                strokeBrush,
                (float)Stroke.Width,
                strokeStyle
            );
        }
        else
        {
            // Draw regular line
            RenderTarget.DrawLine(
                new SharpDX.Vector2((float)startPoint.X, (float)startPoint.Y),
                new SharpDX.Vector2((float)endPoint.X, (float)endPoint.Y),
                strokeBrush,
                (float)Stroke.Width,
                strokeStyle
            );
        }

        // Draw price label
        if (ShowPriceLabel)
        {
            double price = chartScale.GetValueByY((float)startPoint.Y);
            string priceText = price.ToString("0.00");

            // Create text format
            SharpDX.DirectWrite.TextFormat textFormat = new SharpDX.DirectWrite.
                Core.Globals.DirectWriteFactory,
                "Arial",
                10
            );

            // Create text layout
            SharpDX.DirectWrite.TextLayout textLayout = new SharpDX.DirectWrite.
                Core.Globals.DirectWriteFactory,
                priceText,
                textFormat,
                100,
                20
            );

            // Draw text background
            RenderTarget.FillRectangle(
```

```
                new SharpDX.RectangleF((float)chartControl.ActualWidth - 60, (fl
                chartControl.ChartPanel.BarBrushes[Brushes.LightGray]
            );

            // Draw text
            RenderTarget.DrawText(
                priceText,
                textFormat,
                new SharpDX.RectangleF((float)chartControl.ActualWidth - 55, (fl
                chartControl.ChartPanel.BarBrushes[Brushes.Black]
            );

            // Dispose resources
            textLayout.Dispose();
            textFormat.Dispose();
        }
    }

    // Mouse interaction methods
    // (Implementation similar to previous example)

    // Properties
    [XmlIgnore]
    [Display(Name = "Line Style", Order = 1, GroupName = "Visual")]
    public LineStyle LineStyle { get; set; }

    [Display(Name = "Extended Line", Order = 2, GroupName = "Visual")]
    public bool IsExtendedLine { get; set; }

    [Display(Name = "Show Price Label", Order = 3, GroupName = "Visual")]
    public bool ShowPriceLabel { get; set; }

    [XmlIgnore]
    [Display(Name = "Stroke", Order = 4, GroupName = "Visual")]
    public Stroke Stroke { get; set; }

    [Browsable(false)]
    public string StrokeSerializable
    {
        get { return Stroke.ToString(); }
        set { Stroke = new Stroke(value); }
    }

    // Enum for line styles
    public enum LineStyle
    {
        Solid,
        Dashed,
        Dotted
    }
}
```

Price Channel Drawing Tool

```
public class PriceChannelTool : DrawingTool
{
    private ChartAnchor startAnchor;
    private ChartAnchor endAnchor;
    private ChartAnchor widthAnchor;

    public PriceChannelTool()
    {
        startAnchor = new ChartAnchor();
        endAnchor = new ChartAnchor();
        widthAnchor = new ChartAnchor();

        // Set default properties
        ChannelBrush = Brushes.DodgerBlue;
        ChannelOpacity = 20;
        Stroke = new Stroke(Brushes.DodgerBlue, 1);
    }

    public override void OnRender(ChartControl chartControl, ChartScale chartSca
    {
        // Get device points
        Point startPoint = startAnchor.GetPoint(chartControl, chartScale);
        Point endPoint = endAnchor.GetPoint(chartControl, chartScale);
        Point widthPoint = widthAnchor.GetPoint(chartControl, chartScale);

        // Calculate channel width
        double channelWidth = Math.Abs(startPoint.Y - widthPoint.Y);

        // Get brushes
        SharpDX.Direct2D1.Brush strokeBrush = chartControl.ChartPanel.BarBrushes
            chartControl.ChartPanel.BarBrushes[Stroke.BrushDX] :
            chartControl.ChartPanel.BarBrushes.Values.FirstOrDefault();

        SharpDX.Direct2D1.Brush fillBrush = chartControl.ChartPanel.BarBrushes.C
            chartControl.ChartPanel.BarBrushes[ChannelBrush] :
            chartControl.ChartPanel.BarBrushes.Values.FirstOrDefault();

        // Calculate slope and points for the channel
        float slope = (float)(endPoint.Y - startPoint.Y) / (float)(endPoint.X -

        // Upper line points
        float upperStartY = (float)Math.Min(startPoint.Y, widthPoint.Y);
        float upperEndY = upperStartY + slope * ((float)endPoint.X - (float)star

        // Lower line points
        float lowerStartY = (float)Math.Max(startPoint.Y, widthPoint.Y);
```

```
        float lowerEndY = lowerStartY + slope * ((float)endPoint.X - (float)star

        // Create geometry for the channel
        SharpDX.Direct2D1.PathGeometry pathGeometry = new SharpDX.Direct2D1.Path
        SharpDX.Direct2D1.GeometrySink sink = pathGeometry.Open();

        sink.BeginFigure(
            new SharpDX.Vector2((float)startPoint.X, upperStartY),
            SharpDX.Direct2D1.FigureBegin.Filled
        );

        sink.AddLine(new SharpDX.Vector2((float)endPoint.X, upperEndY));
        sink.AddLine(new SharpDX.Vector2((float)endPoint.X, lowerEndY));
        sink.AddLine(new SharpDX.Vector2((float)startPoint.X, lowerStartY));

        sink.EndFigure(SharpDX.Direct2D1.FigureEnd.Closed);
        sink.Close();

        // Draw the filled channel
        RenderTarget.FillGeometry(pathGeometry, fillBrush, null);

        // Draw the upper and lower lines
        RenderTarget.DrawLine(
            new SharpDX.Vector2((float)startPoint.X, upperStartY),
            new SharpDX.Vector2((float)endPoint.X, upperEndY),
            strokeBrush,
            (float)Stroke.Width,
            Stroke.StrokeStyle
        );

        RenderTarget.DrawLine(
            new SharpDX.Vector2((float)startPoint.X, lowerStartY),
            new SharpDX.Vector2((float)endPoint.X, lowerEndY),
            strokeBrush,
            (float)Stroke.Width,
            Stroke.StrokeStyle
        );

        // Draw the middle line
        float middleStartY = (upperStartY + lowerStartY) / 2;
        float middleEndY = (upperEndY + lowerEndY) / 2;

        RenderTarget.DrawLine(
            new SharpDX.Vector2((float)startPoint.X, middleStartY),
            new SharpDX.Vector2((float)endPoint.X, middleEndY),
            strokeBrush,
            (float)Stroke.Width,
            Stroke.StrokeStyle
        );

        // Dispose resources
```

```
            pathGeometry.Dispose();
        }

        // Mouse interaction methods
        // (Implementation similar to previous examples)

        // Properties
        [XmlIgnore]
        [Display(Name = "Channel Brush", Order = 1, GroupName = "Visual")]
        public Brush ChannelBrush { get; set; }

        [Browsable(false)]
        public string ChannelBrushSerializable
        {
            get { return Serialize.BrushToString(ChannelBrush); }
            set { ChannelBrush = Serialize.StringToBrush(value); }
        }

        [Display(Name = "Channel Opacity", Order = 2, GroupName = "Visual")]
        [Range(0, 100)]
        public int ChannelOpacity { get; set; }

        [XmlIgnore]
        [Display(Name = "Stroke", Order = 3, GroupName = "Visual")]
        public Stroke Stroke { get; set; }

        [Browsable(false)]
        public string StrokeSerializable
        {
            get { return Stroke.ToString(); }
            set { Stroke = new Stroke(value); }
        }
    }
```

By understanding these concepts and examples, you can create custom drawing tools to enhance your chart analysis and visualization capabilities in NinjaTrader.

NinjaTrader Candlestick Patterns

This comprehensive guide documents all aspects of candlestick patterns in NinjaTrader, including pattern detection, implementation, and trading strategies.

Table of Contents

- Introduction to Candlestick Patterns
- Candlestick Pattern Types
- NinjaTrader's CandleStickPattern Method

- Implementing Candlestick Pattern Detection
- Custom Candlestick Pattern Detection
- Trading Strategies Based on Candlestick Patterns
- Combining Candlestick Patterns with Other Indicators
- Backtesting Candlestick Pattern Strategies
- Common Issues and Troubleshooting

Introduction to Candlestick Patterns

Candlestick patterns are specific formations on price charts that traders use to identify potential market reversals or continuations. Originating in Japan in the 18th century, these patterns have become a fundamental tool in technical analysis.

History and Significance

Candlestick charting was developed by Japanese rice trader Munehisa Homma in the 1700s. These patterns were introduced to the Western world by Steve Nison in his 1991 book "Japanese Candlestick Charting Techniques." Candlestick patterns provide valuable insights into market psychology and potential price movements.

Anatomy of a Candlestick

A candlestick consists of: - **Body**: The rectangular area between the open and close prices - **Wicks/Shadows**: The thin lines extending above and below the body - **Color**: Typically, a bullish candle (close > open) is white or green, while a bearish candle (close < open) is black or red

Why Candlestick Patterns Matter

Candlestick patterns offer several advantages: - Visual representation of price action - Insight into market psychology - Early warning signals for potential reversals - Identification of continuation patterns - Compatibility with other technical analysis tools

Candlestick Pattern Types

NinjaTrader supports numerous candlestick patterns, which can be categorized into single-candle patterns, double-candle patterns, and multi-candle patterns.

Single-Candle Patterns

Doji

Description: A candle with a very small body, where the open and close prices are nearly equal.

Characteristics: - Tiny or nonexistent body - Open and close prices are very close or equal - Can have long upper and/or lower shadows

Market Psychology: Represents indecision in the market, with neither buyers nor sellers

gaining control.

Code Implementation:

```
// Detecting a Doji pattern
if (CandleStickPattern(ChartPattern.Doji, 4)[0] == 1)
{
    Print("Doji pattern detected at bar " + CurrentBar);
}
```

Visual Identification:

```
    |
    |
----+----
    |
    |
```

Hammer

Description: A bullish reversal pattern that forms during a downtrend, with a small body at the upper end and a long lower shadow.

Characteristics: - Small body at the upper end of the trading range - Long lower shadow (at least twice the length of the body) - Little or no upper shadow - Body color is not important, but a bullish (green/white) body strengthens the signal

Market Psychology: After a downtrend, sellers push prices lower, but buyers step in and push prices back up, indicating potential bullish reversal.

Code Implementation:

```
// Detecting a Hammer pattern
if (CandleStickPattern(ChartPattern.Hammer, 4)[0] == 1)
{
    Print("Hammer pattern detected at bar " + CurrentBar);
}
```

Visual Identification:

```
    _
    |_|
    |
    |
    |
```

Shooting Star

Description: A bearish reversal pattern that forms during an uptrend, with a small body at the lower end and a long upper shadow.

Characteristics: - Small body at the lower end of the trading range - Long upper shadow (at least twice the length of the body) - Little or no lower shadow - Body color is not important, but a bearish (red/black) body strengthens the signal

Market Psychology: After an uptrend, buyers push prices higher, but sellers step in and push prices back down, indicating potential bearish reversal.

Code Implementation:

```
// Detecting a Shooting Star pattern
if (CandleStickPattern(ChartPattern.ShootingStar, 4)[0] == 1)
{
    Print("Shooting Star pattern detected at bar " + CurrentBar);
}
```

Visual Identification:

```
    |
    |
    |
    _
    |_|
```

Inverted Hammer

Description: A bullish reversal pattern that forms during a downtrend, with a small body at the lower end and a long upper shadow.

Characteristics: - Small body at the lower end of the trading range - Long upper shadow (at least twice the length of the body) - Little or no lower shadow - Body color is not important, but a bullish (green/white) body strengthens the signal

Market Psychology: After a downtrend, buyers attempt to push prices higher but fail to maintain the high, yet sellers are unable to push prices lower, indicating potential bullish reversal.

Code Implementation:

```
// Detecting an Inverted Hammer pattern
if (CandleStickPattern(ChartPattern.InvertedHammer, 4)[0] == 1)
{
    Print("Inverted Hammer pattern detected at bar " + CurrentBar);
}
```

Visual Identification:

```
    |
    |
    |
  _|_
   |_|
```

Hanging Man

Description: A bearish reversal pattern that forms during an uptrend, with a small body at the upper end and a long lower shadow.

Characteristics: - Small body at the upper end of the trading range - Long lower shadow (at least twice the length of the body) - Little or no upper shadow - Body color is not important, but a bearish (red/black) body strengthens the signal

Market Psychology: After an uptrend, sellers push prices lower, but buyers step in to push prices back up, yet the failure to make new highs indicates potential bearish reversal.

Code Implementation:

```
// Detecting a Hanging Man pattern
if (CandleStickPattern(ChartPattern.HangingMan, 4)[0] == 1)
{
    Print("Hanging Man pattern detected at bar " + CurrentBar);
}
```

Visual Identification:

```
  _|_
   |_|
    |
    |
    |
```

Marubozu

Description: A candlestick with no shadows, where the open is at the low and the close is at the high (bullish), or the open is at the high and the close is at the low (bearish).

Characteristics: - No upper or lower shadows - Long body - Open is at the low and close is at the high (bullish) - Open is at the high and close is at the low (bearish)

Market Psychology: Represents strong conviction by either buyers (bullish) or sellers (bearish).

Code Implementation:

```
// Detecting a Bullish Marubozu
bool isBullishMarubozu = Close[0] > Open[0] &&
```

```
                          High[0] == Close[0] &&
                          Low[0] == Open[0];

// Detecting a Bearish Marubozu
bool isBearishMarubozu = Close[0] < Open[0] &&
                          High[0] == Open[0] &&
                          Low[0] == Close[0];
```

Visual Identification:

```
Bullish:    Bearish:
 _____       _____
|     |     |     |
|     |     |     |
|_____|     |_____|
```

Double-Candle Patterns

Bullish Engulfing

Description: A bullish reversal pattern that forms during a downtrend, where a bullish candle completely engulfs the previous bearish candle.

Characteristics: - First candle is bearish (red/black) - Second candle is bullish (green/white) - Second candle's body completely engulfs the first candle's body - The pattern is more significant if the second candle also engulfs the shadows

Market Psychology: After a downtrend, buyers overwhelm sellers, indicating a potential bullish reversal.

Code Implementation:

```
// Detecting a Bullish Engulfing pattern
if (CandleStickPattern(ChartPattern.BullishEngulfing, 4)[0] == 1)
{
    Print("Bullish Engulfing pattern detected at bar " + CurrentBar);
}
```

Visual Identification:

```
    |      |
    |     _|_
    |_|   | |
    |_|   | |
    |     |_|
    |      |
```

Bearish Engulfing

Description: A bearish reversal pattern that forms during an uptrend, where a bearish candle completely engulfs the previous bullish candle.

Characteristics: - First candle is bullish (green/white) - Second candle is bearish (red/black) - Second candle's body completely engulfs the first candle's body - The pattern is more significant if the second candle also engulfs the shadows

Market Psychology: After an uptrend, sellers overwhelm buyers, indicating a potential bearish reversal.

Code Implementation:

```
// Detecting a Bearish Engulfing pattern
if (CandleStickPattern(ChartPattern.BearishEngulfing, 4)[0] == 1)
{
    Print("Bearish Engulfing pattern detected at bar " + CurrentBar);
}
```

Visual Identification:

```
    |      |
   _|_     |
   | |    _|_
   | |    | |
   |_|    |_|
    |      |
```

Bullish Harami

Description: A bullish reversal pattern that forms during a downtrend, where a small bullish candle is contained within the body of the previous larger bearish candle.

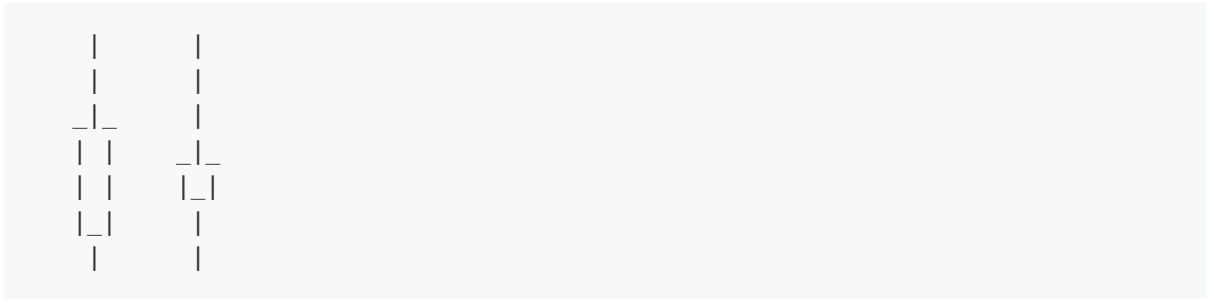
Characteristics: - First candle is bearish (red/black) with a large body - Second candle is bullish (green/white) with a small body - Second candle's body is completely contained within the first candle's body

Market Psychology: After a downtrend, the smaller bullish candle indicates indecision and potential reversal.

Code Implementation:

```
// Detecting a Bullish Harami pattern
if (CandleStickPattern(ChartPattern.BullishHarami, 4)[0] == 1)
{
    Print("Bullish Harami pattern detected at bar " + CurrentBar);
}
```

Visual Identification:



Bearish Harami

Description: A bearish reversal pattern that forms during an uptrend, where a small bearish candle is contained within the body of the previous larger bullish candle.

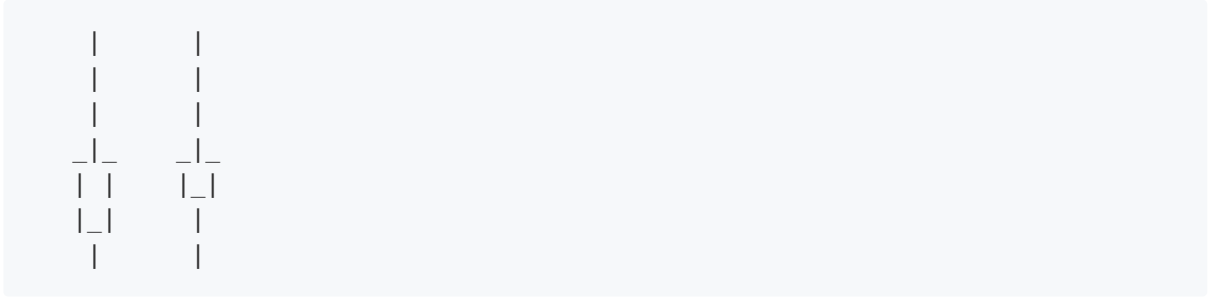
Characteristics: - First candle is bullish (green/white) with a large body - Second candle is bearish (red/black) with a small body - Second candle's body is completely contained within the first candle's body

Market Psychology: After an uptrend, the smaller bearish candle indicates indecision and potential reversal.

Code Implementation:

```
// Detecting a Bearish Harami pattern
if (CandleStickPattern(ChartPattern.BearishHarami, 4)[0] == 1)
{
    Print("Bearish Harami pattern detected at bar " + CurrentBar);
}
```

Visual Identification:



Tweezer Tops

Description: A bearish reversal pattern that forms during an uptrend, where two consecutive candles have identical highs.

Characteristics: - Forms during an uptrend - Two consecutive candles with identical or nearly identical highs - First candle is typically bullish (green/white) - Second candle is typically bearish (red/black)

Market Psychology: After an uptrend, the failure to make a new high on the second candle indicates resistance and potential reversal.

Code Implementation:

```
// Detecting a Tweezer Top pattern
bool isTweezerTop = High[1] == High[0] &&
                    Close[1] > Open[1] &&
                    Close[0] < Open[0] &&
                    IsUptrend(10);

private bool IsUptrend(int lookback)
{
    return Close[0] > SMA(Close, lookback)[0];
}
```

Visual Identification:

```
    |      |
    |      |
   _|_    _|_
   | |    | |
   |_|    |_|
    |      |
```

Tweezer Bottoms

Description: A bullish reversal pattern that forms during a downtrend, where two consecutive candles have identical lows.

Characteristics: - Forms during a downtrend - Two consecutive candles with identical or nearly identical lows - First candle is typically bearish (red/black) - Second candle is typically bullish (green/white)

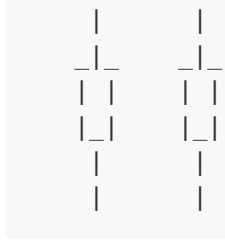
Market Psychology: After a downtrend, the failure to make a new low on the second candle indicates support and potential reversal.

Code Implementation:

```
// Detecting a Tweezer Bottom pattern
bool isTweezerBottom = Low[1] == Low[0] &&
                       Close[1] < Open[1] &&
                       Close[0] > Open[0] &&
                       IsDowntrend(10);

private bool IsDowntrend(int lookback)
{
    return Close[0] < SMA(Close, lookback)[0];
}
```

Visual Identification:



Multi-Candle Patterns

Morning Star

Description: A bullish reversal pattern that forms during a downtrend, consisting of three candles: a large bearish candle, a small-bodied candle, and a large bullish candle.

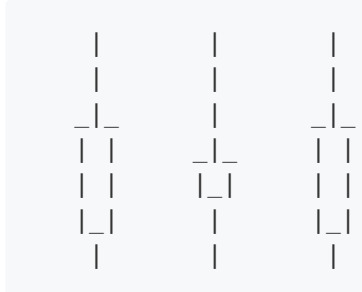
Characteristics: - First candle is bearish (red/black) with a large body - Second candle has a small body and gaps down from the first candle - Third candle is bullish (green/white) with a large body that closes at least halfway into the first candle's body

Market Psychology: After a downtrend, the small-bodied middle candle indicates indecision, followed by strong buying pressure in the third candle, signaling a potential bullish reversal.

Code Implementation:

```
// Detecting a Morning Star pattern
if (CandleStickPattern(ChartPattern.MorningStar, 4)[0] == 1)
{
    Print("Morning Star pattern detected at bar " + CurrentBar);
}
```

Visual Identification:



Evening Star

Description: A bearish reversal pattern that forms during an uptrend, consisting of three candles: a large bullish candle, a small-bodied candle, and a large bearish candle.

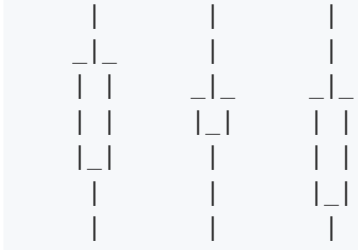
Characteristics: - First candle is bullish (green/white) with a large body - Second candle has a small body and gaps up from the first candle - Third candle is bearish (red/black) with a large body that closes at least halfway into the first candle's body

Market Psychology: After an uptrend, the small-bodied middle candle indicates indecision, followed by strong selling pressure in the third candle, signaling a potential bearish reversal.

Code Implementation:

```
// Detecting an Evening Star pattern
if (CandleStickPattern(ChartPattern.EveningStar, 4)[0] == 1)
{
    Print("Evening Star pattern detected at bar " + CurrentBar);
}
```

Visual Identification:



Three White Soldiers

Description: A bullish reversal pattern that forms during a downtrend, consisting of three consecutive bullish candles, each closing higher than the previous.

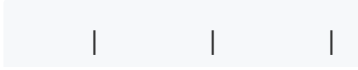
Characteristics: - Three consecutive bullish (green/white) candles - Each candle opens within the body of the previous candle - Each candle closes higher than the previous candle - Each candle has small or no upper shadows

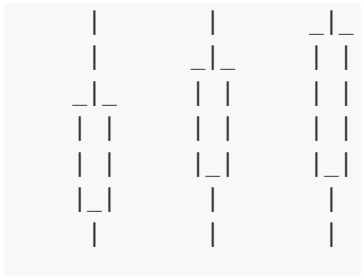
Market Psychology: After a downtrend, three consecutive bullish candles indicate strong buying pressure and a potential bullish reversal.

Code Implementation:

```
// Detecting a Three White Soldiers pattern
if (CandleStickPattern(ChartPattern.RisingThreeMethods, 4)[0] == 1)
{
    Print("Three White Soldiers pattern detected at bar " + CurrentBar);
}
```

Visual Identification:





Three Black Crows

Description: A bearish reversal pattern that forms during an uptrend, consisting of three consecutive bearish candles, each closing lower than the previous.

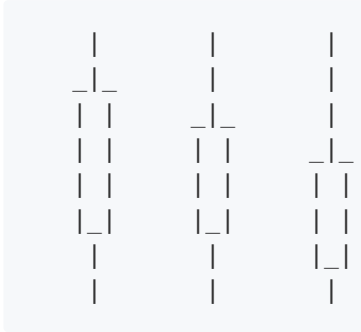
Characteristics: - Three consecutive bearish (red/black) candles - Each candle opens within the body of the previous candle - Each candle closes lower than the previous candle - Each candle has small or no lower shadows

Market Psychology: After an uptrend, three consecutive bearish candles indicate strong selling pressure and a potential bearish reversal.

Code Implementation:

```
// Detecting a Three Black Crows pattern
if (CandleStickPattern(ChartPattern.FallingThreeMethods, 4)[0] == 1)
{
    Print("Three Black Crows pattern detected at bar " + CurrentBar);
}
```

Visual Identification:



Bullish Three Line Strike

Description: A bullish continuation pattern consisting of three bullish candles followed by a bearish candle that engulfs all three bullish candles.

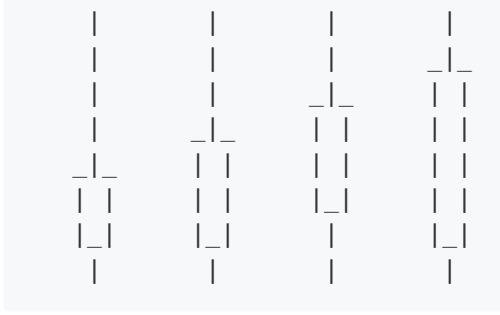
Characteristics: - Three consecutive bullish (green/white) candles, each closing higher than the previous - Fourth candle is bearish (red/black) and opens above the third candle's high - Fourth candle closes below the first candle's open

Market Psychology: Despite the bearish fourth candle, this pattern often leads to a continuation of the bullish trend after a brief pullback.

Code Implementation:

```
// Detecting a Bullish Three Line Strike pattern
bool isBullishThreeLineStrike =
    Close[3] > Open[3] &&
    Close[2] > Open[2] && Close[2] > Close[3] &&
    Close[1] > Open[1] && Close[1] > Close[2] &&
    Open[0] > Close[1] && Close[0] < Open[3];
```

Visual Identification:



Bearish Three Line Strike

Description: A bearish continuation pattern consisting of three bearish candles followed by a bullish candle that engulfs all three bearish candles.

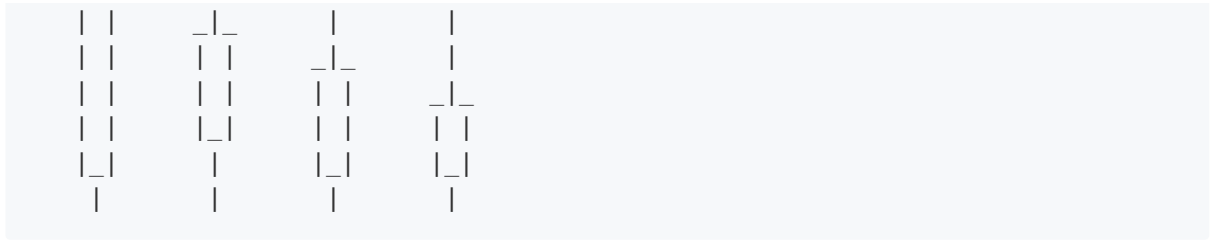
Characteristics: - Three consecutive bearish (red/black) candles, each closing lower than the previous - Fourth candle is bullish (green/white) and opens below the third candle's low - Fourth candle closes above the first candle's open

Market Psychology: Despite the bullish fourth candle, this pattern often leads to a continuation of the bearish trend after a brief rally.

Code Implementation:

```
// Detecting a Bearish Three Line Strike pattern
bool isBearishThreeLineStrike =
    Close[3] < Open[3] &&
    Close[2] < Open[2] && Close[2] < Close[3] &&
    Close[1] < Open[1] && Close[1] < Close[2] &&
    Open[0] < Close[1] && Close[0] > Open[3];
```

Visual Identification:



NinjaTrader's CandleStickPattern Method

NinjaTrader provides a built-in method for detecting candlestick patterns, which simplifies the implementation of pattern-based strategies.

Syntax and Parameters

The CandleStickPattern method has the following syntax:

```
CandleStickPattern(ChartPattern pattern, int trendStrength)
```

Parameters: - **pattern**: The candlestick pattern to detect (from the ChartPattern enumeration) - **trendStrength**: The minimum strength of the trend required for pattern validation (1-10, with 10 being the strongest)

Available Patterns

NinjaTrader supports the following candlestick patterns through the ChartPattern enumeration:

- **BearishBeltHold**: A bearish reversal pattern with a long bearish candle
- **BearishEngulfing**: A bearish reversal pattern where a bearish candle engulfs the previous bullish candle
- **BearishHarami**: A bearish reversal pattern where a small bearish candle is contained within the previous bullish candle
- **BearishHaramiCross**: A bearish harami pattern where the second candle is a doji
- **BullishBeltHold**: A bullish reversal pattern with a long bullish candle
- **BullishEngulfing**: A bullish reversal pattern where a bullish candle engulfs the previous bearish candle
- **BullishHarami**: A bullish reversal pattern where a small bullish candle is contained within the previous bearish candle
- **BullishHaramiCross**: A bullish harami pattern where the second candle is a doji
- **DarkCloudCover**: A bearish reversal pattern where a bearish candle opens above the previous bullish candle's high and closes below its midpoint
- **Doji**: A candle with a very small body, indicating indecision
- **DownsideTasukiGap**: A bullish continuation pattern with a gap down followed by a bearish candle that closes the gap
- **EveningStar**: A bearish reversal pattern with three candles: bullish, small-bodied, and bearish
- **FallingThreeMethods**: A bearish continuation pattern (Three Black Crows)

- **Hammer**: A bullish reversal pattern with a small body at the top and a long lower shadow
- **HangingMan**: A bearish reversal pattern with a small body at the top and a long lower shadow
- **InvertedHammer**: A bullish reversal pattern with a small body at the bottom and a long upper shadow
- **MorningStar**: A bullish reversal pattern with three candles: bearish, small-bodied, and bullish
- **PiercingLine**: A bullish reversal pattern where a bullish candle opens below the previous bearish candle's low and closes above its midpoint
- **RisingThreeMethods**: A bullish continuation pattern (Three White Soldiers)
- **ShootingStar**: A bearish reversal pattern with a small body at the bottom and a long upper shadow

Return Value

The CandleStickPattern method returns a double value representing whether the pattern was found: - **1**: Pattern was found - **0**: Pattern was not found

Example Usage

```
// Detecting a Bullish Engulfing pattern with a trend strength of 4
if (CandleStickPattern(ChartPattern.BullishEngulfing, 4)[0] == 1)
{
    // Pattern detected at the current bar
    Print("Bullish Engulfing pattern detected at bar " + CurrentBar);

    // Take action (e.g., enter a long position)
    EnterLong();
}
```

Implementing Candlestick Pattern Detection

This section provides examples of how to implement candlestick pattern detection in NinjaScript.

Basic Pattern Detection

```
using System;
using System.Collections.Generic;
using NinjaTrader.Cbi;
using NinjaTrader.Data;
using NinjaTrader.NinjaScript;
using NinjaTrader.Core.FloatingPoint;
using NinjaTrader.NinjaScript.Indicators;

namespace NinjaTrader.NinjaScript.Strategies
{
```

```
    public class CandlestickPatternStrategy : Strategy
    {
        private ChartPattern patternToDetect = ChartPattern.BullishEngulfing;
        private int trendStrength = 4;

        protected override void OnStateChange()
        {
            if (State == State.SetDefaults)
            {
                Description = "Strategy based on candlestick patterns";
                Name = "Candlestick Pattern Strategy";

                // Add parameters
                patternToDetect = ChartPattern.BullishEngulfing;
                trendStrength = 4;
            }
        }

        protected override void OnBarUpdate()
        {
            // Wait for enough bars
            if (CurrentBar < 20)
                return;

            // Detect the specified pattern
            if (CandleStickPattern(patternToDetect, trendStrength)[0] == 1)
            {
                Print("Pattern detected: " + patternToDetect.ToString() + " at b

                // Take action based on the pattern
                if (patternToDetect.ToString().StartsWith("Bullish"))
                {
                    EnterLong();
                }
                else if (patternToDetect.ToString().StartsWith("Bearish"))
                {
                    EnterShort();
                }
            }
        }
    }
}
```

Multiple Pattern Detection

```
using System;
using System.Collections.Generic;
using NinjaTrader.Cbi;
```

```
using NinjaTrader.Data;
using NinjaTrader.NinjaScript;
using NinjaTrader.Core.FloatingPoint;
using NinjaTrader.NinjaScript.Indicators;

namespace NinjaTrader.NinjaScript.Strategies
{
    public class MultiPatternStrategy : Strategy
    {
        private ChartPattern[] bullishPatterns;
        private ChartPattern[] bearishPatterns;
        private int trendStrength = 4;

        protected override void OnStateChange()
        {
            if (State == State.SetDefaults)
            {
                Description = "Strategy based on multiple candlestick patterns";
                Name = "Multi-Pattern Strategy";

                // Define bullish patterns
                bullishPatterns = new ChartPattern[]
                {
                    ChartPattern.BullishEngulfing,
                    ChartPattern.BullishHarami,
                    ChartPattern.Hammer,
                    ChartPattern.MorningStar
                };

                // Define bearish patterns
                bearishPatterns = new ChartPattern[]
                {
                    ChartPattern.BearishEngulfing,
                    ChartPattern.BearishHarami,
                    ChartPattern.ShootingStar,
                    ChartPattern.EveningStar
                };

                trendStrength = 4;
            }
        }

        protected override void OnBarUpdate()
        {
            // Wait for enough bars
            if (CurrentBar < 20)
                return;

            // Check for bullish patterns
            foreach (ChartPattern pattern in bullishPatterns)
            {
```

```
                if (CandleStickPattern(pattern, trendStrength)[0] == 1)
                {
                    Print("Bullish pattern detected: " + pattern.ToString() + "

                    // Enter long if not already in a long position
                    if (Position.MarketPosition != MarketPosition.Long)
                    {
                        EnterLong();
                    }

                    break; // Only act on the first pattern found
                }
            }

            // Check for bearish patterns
            foreach (ChartPattern pattern in bearishPatterns)
            {
                if (CandleStickPattern(pattern, trendStrength)[0] == 1)
                {
                    Print("Bearish pattern detected: " + pattern.ToString() + "

                    // Enter short if not already in a short position
                    if (Position.MarketPosition != MarketPosition.Short)
                    {
                        EnterShort();
                    }

                    break; // Only act on the first pattern found
                }
            }
        }
    }
}
```

Pattern Detection with Confirmation

```
using System;
using System.Collections.Generic;
using NinjaTrader.Cbi;
using NinjaTrader.Data;
using NinjaTrader.NinjaScript;
using NinjaTrader.Core.FloatingPoint;
using NinjaTrader.NinjaScript.Indicators;

namespace NinjaTrader.NinjaScript.Strategies
{
    public class PatternWithConfirmationStrategy : Strategy
    {
```

```
        private RSI rsi;
        private ChartPattern[] bullishPatterns;
        private ChartPattern[] bearishPatterns;
        private int trendStrength = 4;

        protected override void OnStateChange()
        {
            if (State == State.Configure)
            {
                // Add RSI indicator for confirmation
                rsi = RSI(14);

                // Define bullish patterns
                bullishPatterns = new ChartPattern[]
                {
                    ChartPattern.BullishEngulfing,
                    ChartPattern.BullishHarami,
                    ChartPattern.Hammer,
                    ChartPattern.MorningStar
                };

                // Define bearish patterns
                bearishPatterns = new ChartPattern[]
                {
                    ChartPattern.BearishEngulfing,
                    ChartPattern.BearishHarami,
                    ChartPattern.ShootingStar,
                    ChartPattern.EveningStar
                };
            }
        }

        protected override void OnBarUpdate()
        {
            // Wait for enough bars
            if (CurrentBar < 20)
                return;

            // Check for bullish patterns with RSI confirmation
            foreach (ChartPattern pattern in bullishPatterns)
            {
                if (CandleStickPattern(pattern, trendStrength)[0] == 1 && rsi[0]
                {
                    Print("Confirmed bullish pattern: " + pattern.ToString() + "

                    // Enter long if not already in a long position
                    if (Position.MarketPosition != MarketPosition.Long)
                    {
                        EnterLong();
                    }
```

```
                    break; // Only act on the first pattern found
                }
            }

            // Check for bearish patterns with RSI confirmation
            foreach (ChartPattern pattern in bearishPatterns)
            {
                if (CandleStickPattern(pattern, trendStrength)[0] == 1 && rsi[0]
                {
                    Print("Confirmed bearish pattern: " + pattern.ToString() + "

                    // Enter short if not already in a short position
                    if (Position.MarketPosition != MarketPosition.Short)
                    {
                        EnterShort();
                    }

                    break; // Only act on the first pattern found
                }
            }
        }
    }
}
```

Custom Candlestick Pattern Detection

While NinjaTrader provides built-in detection for many common candlestick patterns, you may want to create custom pattern detection for patterns not included or to implement more specific criteria.

Creating a Custom Pattern Detector

```
using System;
using System.Collections.Generic;
using NinjaTrader.Cbi;
using NinjaTrader.Data;
using NinjaTrader.NinjaScript;
using NinjaTrader.Core.FloatingPoint;
using NinjaTrader.NinjaScript.Indicators;

namespace NinjaTrader.NinjaScript.Indicators
{
    public class CustomCandlestickPatterns : Indicator
    {
        private Series<double> bullishPinBar;
        private Series<double> bearishPinBar;
```

```
        protected override void OnStateChange()
        {
            if (State == State.SetDefaults)
            {
                Description = "Custom Candlestick Pattern Detector";
                Name = "CustomCandlestickPatterns";
                Calculate = Calculate.OnBarClose;
                IsOverlay = true;
            }
            else if (State == State.DataLoaded)
            {
                // Create series for pattern detection results
                bullishPinBar = new Series<double>(this);
                bearishPinBar = new Series<double>(this);
            }
        }

        protected override void OnBarUpdate()
        {
            // Detect Bullish Pin Bar (Hammer-like pattern with specific criteri
            bullishPinBar[0] = IsBullishPinBar() ? 1 : 0;

            // Detect Bearish Pin Bar (Shooting Star-like pattern with specific
            bearishPinBar[0] = IsBearishPinBar() ? 1 : 0;

            // Draw arrows for detected patterns
            if (bullishPinBar[0] == 1)
            {
                Draw.ArrowUp(this, "BullishPinBar" + CurrentBar, false, 0, Low[0
            }

            if (bearishPinBar[0] == 1)
            {
                Draw.ArrowDown(this, "BearishPinBar" + CurrentBar, false, 0, Hig
            }
        }

        // Custom method to detect Bullish Pin Bar
        private bool IsBullishPinBar()
        {
            // Calculate candle parts
            double body = Math.Abs(Close[0] - Open[0]);
            double upperWick = High[0] - Math.Max(Open[0], Close[0]);
            double lowerWick = Math.Min(Open[0], Close[0]) - Low[0];
            double totalRange = High[0] - Low[0];

            // Pin Bar criteria
            bool hasLongLowerWick = lowerWick > body * 2 && lowerWick > upperWic
            bool hasSmallBody = body < totalRange * 0.3;
            bool isInDowntrend = Close[1] < Close[5] && Close[2] < Close[6];
```

```
            return hasLongLowerWick && hasSmallBody && isInDowntrend;
        }

        // Custom method to detect Bearish Pin Bar
        private bool IsBearishPinBar()
        {
            // Calculate candle parts
            double body = Math.Abs(Close[0] - Open[0]);
            double upperWick = High[0] - Math.Max(Open[0], Close[0]);
            double lowerWick = Math.Min(Open[0], Close[0]) - Low[0];
            double totalRange = High[0] - Low[0];

            // Pin Bar criteria
            bool hasLongUpperWick = upperWick > body * 2 && upperWick > lowerWic
            bool hasSmallBody = body < totalRange * 0.3;
            bool isInUptrend = Close[1] > Close[5] && Close[2] > Close[6];

            return hasLongUpperWick && hasSmallBody && isInUptrend;
        }
    }
}
```

Using Custom Pattern Detection in a Strategy

```
using System;
using System.Collections.Generic;
using NinjaTrader.Cbi;
using NinjaTrader.Data;
using NinjaTrader.NinjaScript;
using NinjaTrader.Core.FloatingPoint;
using NinjaTrader.NinjaScript.Indicators;

namespace NinjaTrader.NinjaScript.Strategies
{
    public class CustomPatternStrategy : Strategy
    {
        private CustomCandlestickPatterns customPatterns;

        protected override void OnStateChange()
        {
            if (State == State.Configure)
            {
                // Add custom pattern detector
                customPatterns = CustomCandlestickPatterns();
            }
        }

        protected override void OnBarUpdate()
```

```
        {
            // Wait for enough bars
            if (CurrentBar < 20)
                return;

            // Check for bullish pin bar
            if (customPatterns.BullishPinBar[0] == 1)
            {
                Print("Bullish Pin Bar detected at bar " + CurrentBar);

                // Enter long if not already in a long position
                if (Position.MarketPosition != MarketPosition.Long)
                {
                    EnterLong();
                }
            }

            // Check for bearish pin bar
            if (customPatterns.BearishPinBar[0] == 1)
            {
                Print("Bearish Pin Bar detected at bar " + CurrentBar);

                // Enter short if not already in a short position
                if (Position.MarketPosition != MarketPosition.Short)
                {
                    EnterShort();
                }
            }
        }
    }
}
```

Trading Strategies Based on Candlestick Patterns

This section provides examples of complete trading strategies based on candlestick patterns.

Reversal Strategy

```
using System;
using System.Collections.Generic;
using NinjaTrader.Cbi;
using NinjaTrader.Data;
using NinjaTrader.NinjaScript;
using NinjaTrader.Core.FloatingPoint;
using NinjaTrader.NinjaScript.Indicators;

namespace NinjaTrader.NinjaScript.Strategies
{
```

```
public class CandlestickReversalStrategy : Strategy
{
    private SMA fastSMA;
    private SMA slowSMA;

    protected override void OnStateChange()
    {
        if (State == State.SetDefaults)
        {
            Description = "Strategy based on candlestick reversal patterns";
            Name = "Candlestick Reversal Strategy";
        }
        else if (State == State.Configure)
        {
            // Add indicators for trend determination
            fastSMA = SMA(10);
            slowSMA = SMA(30);
        }
    }

    protected override void OnBarUpdate()
    {
        // Wait for enough bars
        if (CurrentBar < 30)
            return;

        // Determine trend
        bool isUptrend = fastSMA[0] > slowSMA[0];
        bool isDowntrend = fastSMA[0] < slowSMA[0];

        // Look for reversal patterns in downtrend
        if (isDowntrend)
        {
            // Check for bullish reversal patterns
            if (CandleStickPattern(ChartPattern.BullishEngulfing, 4)[0] == 1
                CandleStickPattern(ChartPattern.Hammer, 4)[0] == 1 ||
                CandleStickPattern(ChartPattern.MorningStar, 4)[0] == 1)
            {
                Print("Bullish reversal pattern detected in downtrend at bar

                // Enter long with stop loss and profit target
                EnterLong();
                SetStopLoss(CalculationMode.Ticks, 20);
                SetProfitTarget(CalculationMode.Ticks, 40);
            }
        }

        // Look for reversal patterns in uptrend
        if (isUptrend)
        {
            // Check for bearish reversal patterns
```

```
                if (CandleStickPattern(ChartPattern.BearishEngulfing, 4)[0] == 1
                    CandleStickPattern(ChartPattern.ShootingStar, 4)[0] == 1 ||
                    CandleStickPattern(ChartPattern.EveningStar, 4)[0] == 1)
                {
                    Print("Bearish reversal pattern detected in uptrend at bar "

                    // Enter short with stop loss and profit target
                    EnterShort();
                    SetStopLoss(CalculationMode.Ticks, 20);
                    SetProfitTarget(CalculationMode.Ticks, 40);
                }
            }
        }
    }
}
```

Continuation Strategy

```
using System;
using System.Collections.Generic;
using NinjaTrader.Cbi;
using NinjaTrader.Data;
using NinjaTrader.NinjaScript;
using NinjaTrader.Core.FloatingPoint;
using NinjaTrader.NinjaScript.Indicators;

namespace NinjaTrader.NinjaScript.Strategies
{
    public class CandlestickContinuationStrategy : Strategy
    {
        private SMA fastSMA;
        private SMA slowSMA;

        protected override void OnStateChange()
        {
            if (State == State.SetDefaults)
            {
                Description = "Strategy based on candlestick continuation patter
                Name = "Candlestick Continuation Strategy";
            }
            else if (State == State.Configure)
            {
                // Add indicators for trend determination
                fastSMA = SMA(10);
                slowSMA = SMA(30);
            }
        }
```

```
        protected override void OnBarUpdate()
        {
            // Wait for enough bars
            if (CurrentBar < 30)
                return;

            // Determine trend
            bool isUptrend = fastSMA[0] > slowSMA[0];
            bool isDowntrend = fastSMA[0] < slowSMA[0];

            // Look for continuation patterns in uptrend
            if (isUptrend)
            {
                // Check for bullish continuation patterns
                if (CandleStickPattern(ChartPattern.RisingThreeMethods, 4)[0] ==
                    IsBullishHarami())
                {
                    Print("Bullish continuation pattern detected in uptrend at b

                    // Enter long with stop loss and profit target
                    EnterLong();
                    SetStopLoss(CalculationMode.Ticks, 15);
                    SetProfitTarget(CalculationMode.Ticks, 30);
                }
            }

            // Look for continuation patterns in downtrend
            if (isDowntrend)
            {
                // Check for bearish continuation patterns
                if (CandleStickPattern(ChartPattern.FallingThreeMethods, 4)[0] =
                    IsBearishHarami())
                {
                    Print("Bearish continuation pattern detected in downtrend at

                    // Enter short with stop loss and profit target
                    EnterShort();
                    SetStopLoss(CalculationMode.Ticks, 15);
                    SetProfitTarget(CalculationMode.Ticks, 30);
                }
            }
        }

        // Custom method to detect Bullish Harami in uptrend (as continuation)
        private bool IsBullishHarami()
        {
            bool isUptrend = fastSMA[0] > slowSMA[0];
            return isUptrend && CandleStickPattern(ChartPattern.BullishHarami, 4
        }

        // Custom method to detect Bearish Harami in downtrend (as continuation)
```

```
        private bool IsBearishHarami()
        {
            bool isDowntrend = fastSMA[0] < slowSMA[0];
            return isDowntrend && CandleStickPattern(ChartPattern.BearishHarami,
        }
    }
}
```

Comprehensive Pattern Strategy

```
using System;
using System.Collections.Generic;
using NinjaTrader.Cbi;
using NinjaTrader.Data;
using NinjaTrader.NinjaScript;
using NinjaTrader.Core.FloatingPoint;
using NinjaTrader.NinjaScript.Indicators;

namespace NinjaTrader.NinjaScript.Strategies
{
    public class ComprehensiveCandlestickStrategy : Strategy
    {
        private SMA fastSMA;
        private SMA slowSMA;
        private RSI rsi;
        private ATR atr;

        private ChartPattern[] bullishReversalPatterns;
        private ChartPattern[] bearishReversalPatterns;
        private ChartPattern[] bullishContinuationPatterns;
        private ChartPattern[] bearishContinuationPatterns;

        protected override void OnStateChange()
        {
            if (State == State.SetDefaults)
            {
                Description = "Comprehensive strategy based on candlestick patte
                Name = "Comprehensive Candlestick Strategy";
            }
            else if (State == State.Configure)
            {
                // Add indicators
                fastSMA = SMA(10);
                slowSMA = SMA(30);
                rsi = RSI(14);
                atr = ATR(14);

                // Define pattern arrays
```

```
            bullishReversalPatterns = new ChartPattern[]
            {
                ChartPattern.BullishEngulfing,
                ChartPattern.Hammer,
                ChartPattern.MorningStar,
                ChartPattern.BullishHarami
            };

            bearishReversalPatterns = new ChartPattern[]
            {
                ChartPattern.BearishEngulfing,
                ChartPattern.ShootingStar,
                ChartPattern.EveningStar,
                ChartPattern.BearishHarami
            };

            bullishContinuationPatterns = new ChartPattern[]
            {
                ChartPattern.RisingThreeMethods
            };

            bearishContinuationPatterns = new ChartPattern[]
            {
                ChartPattern.FallingThreeMethods
            };
        }
    }

    protected override void OnBarUpdate()
    {
        // Wait for enough bars
        if (CurrentBar < 30)
            return;

        // Determine trend
        bool isUptrend = fastSMA[0] > slowSMA[0];
        bool isDowntrend = fastSMA[0] < slowSMA[0];

        // Check for entry signals
        if (Position.MarketPosition == MarketPosition.Flat)
        {
            // Look for bullish reversal patterns in downtrend
            if (isDowntrend && rsi[0] < 30)
            {
                foreach (ChartPattern pattern in bullishReversalPatterns)
                {
                    if (CandleStickPattern(pattern, 4)[0] == 1)
                    {
                        Print("Bullish reversal pattern detected: " + patter

                        // Enter long with dynamic stop loss based on ATR
```

```
                    EnterLong();
                    SetStopLoss(CalculationMode.Price, Low[0] - atr[0] *
                    SetProfitTarget(CalculationMode.Price, Close[0] + at

                    break;
                }
            }
        }

        // Look for bearish reversal patterns in uptrend
        if (isUptrend && rsi[0] > 70)
        {
            foreach (ChartPattern pattern in bearishReversalPatterns)
            {
                if (CandleStickPattern(pattern, 4)[0] == 1)
                {
                    Print("Bearish reversal pattern detected: " + patter

                    // Enter short with dynamic stop loss based on ATR
                    EnterShort();
                    SetStopLoss(CalculationMode.Price, High[0] + atr[0]
                    SetProfitTarget(CalculationMode.Price, Close[0] - at

                    break;
                }
            }
        }

        // Look for bullish continuation patterns in uptrend
        if (isUptrend && rsi[0] > 40 && rsi[0] < 60)
        {
            foreach (ChartPattern pattern in bullishContinuationPatterns
            {
                if (CandleStickPattern(pattern, 4)[0] == 1)
                {
                    Print("Bullish continuation pattern detected: " + pa

                    // Enter long with dynamic stop loss based on ATR
                    EnterLong();
                    SetStopLoss(CalculationMode.Price, Low[0] - atr[0] *
                    SetProfitTarget(CalculationMode.Price, Close[0] + at

                    break;
                }
            }
        }

        // Look for bearish continuation patterns in downtrend
        if (isDowntrend && rsi[0] > 40 && rsi[0] < 60)
        {
            foreach (ChartPattern pattern in bearishContinuationPatterns
```

```
                    {
                        if (CandleStickPattern(pattern, 4)[0] == 1)
                        {
                            Print("Bearish continuation pattern detected: " + pa

                            // Enter short with dynamic stop loss based on ATR
                            EnterShort();
                            SetStopLoss(CalculationMode.Price, High[0] + atr[0]
                            SetProfitTarget(CalculationMode.Price, Close[0] - at

                            break;
                        }
                    }
                }
            }
        }
    }
}
```

Combining Candlestick Patterns with Other Indicators

Candlestick patterns are most effective when combined with other technical indicators for confirmation. This section provides examples of how to combine candlestick patterns with various indicators.

Candlestick Patterns with Moving Averages

```
using System;
using System.Collections.Generic;
using NinjaTrader.Cbi;
using NinjaTrader.Data;
using NinjaTrader.NinjaScript;
using NinjaTrader.Core.FloatingPoint;
using NinjaTrader.NinjaScript.Indicators;

namespace NinjaTrader.NinjaScript.Strategies
{
    public class CandlestickWithMAStrategy : Strategy
    {
        private SMA shortSMA;
        private SMA mediumSMA;
        private SMA longSMA;

        protected override void OnStateChange()
        {
            if (State == State.SetDefaults)
            {
```

```
            Description = "Strategy combining candlestick patterns with movi
            Name = "Candlestick with MA Strategy";
        }
        else if (State == State.Configure)
        {
            // Add moving averages
            shortSMA = SMA(10);
            mediumSMA = SMA(20);
            longSMA = SMA(50);
        }
    }

    protected override void OnBarUpdate()
    {
        // Wait for enough bars
        if (CurrentBar < 50)
            return;

        // Determine trend using moving averages
        bool isStrongUptrend = shortSMA[0] > mediumSMA[0] && mediumSMA[0] >
        bool isStrongDowntrend = shortSMA[0] < mediumSMA[0] && mediumSMA[0]

        // Look for bullish patterns at support (long SMA)
        if (Low[0] <= longSMA[0] && Close[0] > longSMA[0] && isStrongUptrend
        {
            // Check for bullish patterns
            if (CandleStickPattern(ChartPattern.BullishEngulfing, 4)[0] == 1
                CandleStickPattern(ChartPattern.Hammer, 4)[0] == 1)
            {
                Print("Bullish pattern at support detected");

                // Enter long
                EnterLong();
                SetStopLoss(CalculationMode.Price, Math.Min(Low[0], Low[1])
                SetProfitTarget(CalculationMode.Price, Close[0] + (Close[0]
            }
        }

        // Look for bearish patterns at resistance (long SMA)
        if (High[0] >= longSMA[0] && Close[0] < longSMA[0] && isStrongDowntr
        {
            // Check for bearish patterns
            if (CandleStickPattern(ChartPattern.BearishEngulfing, 4)[0] == 1
                CandleStickPattern(ChartPattern.ShootingStar, 4)[0] == 1)
            {
                Print("Bearish pattern at resistance detected");

                // Enter short
                EnterShort();
                SetStopLoss(CalculationMode.Price, Math.Max(High[0], High[1]
                SetProfitTarget(CalculationMode.Price, Close[0] - (longSMA[0
```

```
                }
            }
        }
    }
}
```

Candlestick Patterns with RSI

```
using System;
using System.Collections.Generic;
using NinjaTrader.Cbi;
using NinjaTrader.Data;
using NinjaTrader.NinjaScript;
using NinjaTrader.Core.FloatingPoint;
using NinjaTrader.NinjaScript.Indicators;

namespace NinjaTrader.NinjaScript.Strategies
{
    public class CandlestickWithRSIStrategy : Strategy
    {
        private RSI rsi;

        protected override void OnStateChange()
        {
            if (State == State.SetDefaults)
            {
                Description = "Strategy combining candlestick patterns with RSI"
                Name = "Candlestick with RSI Strategy";
            }
            else if (State == State.Configure)
            {
                // Add RSI indicator
                rsi = RSI(14);
            }
        }

        protected override void OnBarUpdate()
        {
            // Wait for enough bars
            if (CurrentBar < 20)
                return;

            // Look for bullish patterns in oversold conditions
            if (rsi[0] < 30)
            {
                // Check for bullish patterns
                if (CandleStickPattern(ChartPattern.BullishEngulfing, 4)[0] == 1
                    CandleStickPattern(ChartPattern.Hammer, 4)[0] == 1 ||
```

```
                    CandleStickPattern(ChartPattern.MorningStar, 4)[0] == 1)
                {
                    Print("Bullish pattern in oversold condition detected");

                    // Enter long
                    EnterLong();
                    SetStopLoss(CalculationMode.Price, Math.Min(Low[0], Low[1])
                    SetProfitTarget(CalculationMode.Price, Close[0] + (Close[0]
                }
            }

            // Look for bearish patterns in overbought conditions
            if (rsi[0] > 70)
            {
                // Check for bearish patterns
                if (CandleStickPattern(ChartPattern.BearishEngulfing, 4)[0] == 1
                    CandleStickPattern(ChartPattern.ShootingStar, 4)[0] == 1 ||
                    CandleStickPattern(ChartPattern.EveningStar, 4)[0] == 1)
                {
                    Print("Bearish pattern in overbought condition detected");

                    // Enter short
                    EnterShort();
                    SetStopLoss(CalculationMode.Price, Math.Max(High[0], High[1]
                    SetProfitTarget(CalculationMode.Price, Close[0] - (High[0] -
                }
            }
        }
    }
}
```

Candlestick Patterns with Support/Resistance

```
using System;
using System.Collections.Generic;
using NinjaTrader.Cbi;
using NinjaTrader.Data;
using NinjaTrader.NinjaScript;
using NinjaTrader.Core.FloatingPoint;
using NinjaTrader.NinjaScript.Indicators;

namespace NinjaTrader.NinjaScript.Strategies
{
    public class CandlestickWithSupportResistanceStrategy : Strategy
    {
        private double[] supportLevels;
        private double[] resistanceLevels;
        private int lookbackPeriod = 50;
```

```
        protected override void OnStateChange()
        {
            if (State == State.SetDefaults)
            {
                Description = "Strategy combining candlestick patterns with supp
                Name = "Candlestick with Support/Resistance Strategy";
            }
        }

        protected override void OnBarUpdate()
        {
            // Wait for enough bars
            if (CurrentBar < lookbackPeriod)
                return;

            // Update support and resistance levels every 10 bars
            if (CurrentBar % 10 == 0 || supportLevels == null)
            {
                FindSupportResistanceLevels();
            }

            // Check if price is near support level
            double nearestSupport = FindNearestLevel(supportLevels, Close[0]);
            if (Math.Abs(Low[0] - nearestSupport) <= 5 * TickSize)
            {
                // Check for bullish patterns
                if (CandleStickPattern(ChartPattern.BullishEngulfing, 4)[0] == 1
                    CandleStickPattern(ChartPattern.Hammer, 4)[0] == 1)
                {
                    Print("Bullish pattern at support level detected");

                    // Enter long
                    EnterLong();
                    SetStopLoss(CalculationMode.Price, nearestSupport - 10 * Tic
                    SetProfitTarget(CalculationMode.Price, Close[0] + (Close[0]
                }
            }

            // Check if price is near resistance level
            double nearestResistance = FindNearestLevel(resistanceLevels, Close[
            if (Math.Abs(High[0] - nearestResistance) <= 5 * TickSize)
            {
                // Check for bearish patterns
                if (CandleStickPattern(ChartPattern.BearishEngulfing, 4)[0] == 1
                    CandleStickPattern(ChartPattern.ShootingStar, 4)[0] == 1)
                {
                    Print("Bearish pattern at resistance level detected");

                    // Enter short
                    EnterShort();
```

```
                SetStopLoss(CalculationMode.Price, nearestResistance + 10 *
                SetProfitTarget(CalculationMode.Price, Close[0] - (nearestRe
            }
        }
    }

    private void FindSupportResistanceLevels()
    {
        List<double> supports = new List<double>();
        List<double> resistances = new List<double>();

        // Find swing highs and lows
        for (int i = 5; i < lookbackPeriod - 5; i++)
        {
            // Swing low
            if (Low[i] < Low[i-1] && Low[i] < Low[i-2] && Low[i] < Low[i+1]
            {
                supports.Add(Low[i]);
            }

            // Swing high
            if (High[i] > High[i-1] && High[i] > High[i-2] && High[i] > High
            {
                resistances.Add(High[i]);
            }
        }

        // Convert lists to arrays
        supportLevels = supports.ToArray();
        resistanceLevels = resistances.ToArray();
    }

    private double FindNearestLevel(double[] levels, double price)
    {
        if (levels == null || levels.Length == 0)
            return 0;

        double nearestLevel = levels[0];
        double minDistance = Math.Abs(price - levels[0]);

        foreach (double level in levels)
        {
            double distance = Math.Abs(price - level);
            if (distance < minDistance)
            {
                minDistance = distance;
                nearestLevel = level;
            }
        }

        return nearestLevel;
```

```
        }
    }
}
```

Backtesting Candlestick Pattern Strategies

This section provides guidance on backtesting strategies based on candlestick patterns.

Optimizing Pattern Parameters

```
using System;
using System.Collections.Generic;
using NinjaTrader.Cbi;
using NinjaTrader.Data;
using NinjaTrader.NinjaScript;
using NinjaTrader.Core.FloatingPoint;
using NinjaTrader.NinjaScript.Indicators;

namespace NinjaTrader.NinjaScript.Strategies
{
    public class OptimizedCandlestickStrategy : Strategy
    {
        private RSI rsi;

        [NinjaScriptProperty]
        [Range(1, 10)]
        [Display(Name = "Trend Strength", Description = "Strength of trend requi
        public int TrendStrength { get; set; }

        [NinjaScriptProperty]
        [Range(10, 30)]
        [Display(Name = "RSI Oversold", Description = "RSI level for oversold co
        public int RsiOversold { get; set; }

        [NinjaScriptProperty]
        [Range(70, 90)]
        [Display(Name = "RSI Overbought", Description = "RSI level for overbough
        public int RsiOverbought { get; set; }

        [NinjaScriptProperty]
        [Range(10, 30)]
        [Display(Name = "Stop Loss (Ticks)", Description = "Stop loss distance i
        public int StopLossTicks { get; set; }

        [NinjaScriptProperty]
        [Range(20, 60)]
        [Display(Name = "Profit Target (Ticks)", Description = "Profit target di
        public int ProfitTargetTicks { get; set; }
```

```
        protected override void OnStateChange()
        {
            if (State == State.SetDefaults)
            {
                Description = "Optimized strategy based on candlestick patterns"
                Name = "Optimized Candlestick Strategy";

                // Default parameter values
                TrendStrength = 4;
                RsiOversold = 30;
                RsiOverbought = 70;
                StopLossTicks = 15;
                ProfitTargetTicks = 30;
            }
            else if (State == State.Configure)
            {
                // Add RSI indicator
                rsi = RSI(14);
            }
        }

        protected override void OnBarUpdate()
        {
            // Wait for enough bars
            if (CurrentBar < 20)
                return;

            // Look for bullish patterns in oversold conditions
            if (rsi[0] < RsiOversold)
            {
                // Check for bullish patterns with optimized trend strength
                if (CandleStickPattern(ChartPattern.BullishEngulfing, TrendStren
                    CandleStickPattern(ChartPattern.Hammer, TrendStrength)[0] ==
                {
                    Print("Bullish pattern detected with trend strength " + Tren

                    // Enter long with optimized stop loss and profit target
                    EnterLong();
                    SetStopLoss(CalculationMode.Ticks, StopLossTicks);
                    SetProfitTarget(CalculationMode.Ticks, ProfitTargetTicks);
                }
            }

            // Look for bearish patterns in overbought conditions
            if (rsi[0] > RsiOverbought)
            {
                // Check for bearish patterns with optimized trend strength
                if (CandleStickPattern(ChartPattern.BearishEngulfing, TrendStren
                    CandleStickPattern(ChartPattern.ShootingStar, TrendStrength)
                {
```

```
                    Print("Bearish pattern detected with trend strength " + Tren

                    // Enter short with optimized stop loss and profit target
                    EnterShort();
                    SetStopLoss(CalculationMode.Ticks, StopLossTicks);
                    SetProfitTarget(CalculationMode.Ticks, ProfitTargetTicks);
                }
            }
        }
    }
}
```

Analyzing Pattern Performance

```
using System;
using System.Collections.Generic;
using NinjaTrader.Cbi;
using NinjaTrader.Data;
using NinjaTrader.NinjaScript;
using NinjaTrader.Core.FloatingPoint;
using NinjaTrader.NinjaScript.Indicators;

namespace NinjaTrader.NinjaScript.Strategies
{
    public class PatternPerformanceAnalyzer : Strategy
    {
        private Dictionary<string, PatternStats> patternStats;
        private ChartPattern[] patternsToAnalyze;

        protected override void OnStateChange()
        {
            if (State == State.SetDefaults)
            {
                Description = "Analyzes the performance of different candlestick
                Name = "Pattern Performance Analyzer";
            }
            else if (State == State.Configure)
            {
                // Initialize pattern statistics dictionary
                patternStats = new Dictionary<string, PatternStats>();

                // Define patterns to analyze
                patternsToAnalyze = new ChartPattern[]
                {
                    ChartPattern.BullishEngulfing,
                    ChartPattern.BearishEngulfing,
                    ChartPattern.Hammer,
                    ChartPattern.ShootingStar,
```

```
                    ChartPattern.MorningStar,
                    ChartPattern.EveningStar,
                    ChartPattern.Doji
                };

                // Initialize statistics for each pattern
                foreach (ChartPattern pattern in patternsToAnalyze)
                {
                    patternStats[pattern.ToString()] = new PatternStats();
                }
            }
            else if (State == State.Terminated)
            {
                // Print performance statistics
                Print("Pattern Performance Analysis:");
                Print("----------------------------");

                foreach (KeyValuePair<string, PatternStats> kvp in patternStats)
                {
                    string patternName = kvp.Key;
                    PatternStats stats = kvp.Value;

                    if (stats.TotalOccurrences > 0)
                    {
                        double successRate = (double)stats.SuccessfulPredictions
                        double averageReturn = stats.TotalReturn / stats.TotalOc

                        Print(patternName + ":");
                        Print("  Total Occurrences: " + stats.TotalOccurrences);
                        Print("  Successful Predictions: " + stats.SuccessfulPre
                        Print("  Average Return: " + averageReturn.ToString("0.0
                        Print("  Max Return: " + stats.MaxReturn + " ticks");
                        Print("  Min Return: " + stats.MinReturn + " ticks");
                        Print("----------------------------");
                    }
                }
            }
        }

        protected override void OnBarUpdate()
        {
            // Wait for enough bars
            if (CurrentBar < 20)
                return;

            // Analyze each pattern
            foreach (ChartPattern pattern in patternsToAnalyze)
            {
                // Check if pattern is detected
                if (CandleStickPattern(pattern, 4)[0] == 1)
                {
```

```
                    // Record pattern occurrence
                    patternStats[pattern.ToString()].TotalOccurrences++;

                    // Determine expected direction
                    bool isBullish = pattern.ToString().StartsWith("Bullish") ||
                                     pattern == ChartPattern.Hammer ||
                                     pattern == ChartPattern.MorningStar;

                    bool isBearish = pattern.ToString().StartsWith("Bearish") ||
                                     pattern == ChartPattern.ShootingStar ||
                                     pattern == ChartPattern.EveningStar;

                    // For Doji, determine direction based on context
                    if (pattern == ChartPattern.Doji)
                    {
                        if (Close[1] < Open[1]) // Previous bar was bearish
                            isBullish = true;
                        else
                            isBearish = true;
                    }

                    // Record entry price
                    double entryPrice = Close[0];

                    // Look ahead to determine outcome (10 bars)
                    double maxPrice = High[0];
                    double minPrice = Low[0];

                    for (int i = 1; i <= 10 && CurrentBar - i >= 0; i++)
                    {
                        maxPrice = Math.Max(maxPrice, High[i]);
                        minPrice = Math.Min(minPrice, Low[i]);
                    }

                    // Calculate returns
                    double bullishReturn = (maxPrice - entryPrice) / TickSize;
                    double bearishReturn = (entryPrice - minPrice) / TickSize;

                    // Update statistics
                    if (isBullish)
                    {
                        // For bullish patterns, success is defined as price mov
                        if (maxPrice > entryPrice + 5 * TickSize)
                            patternStats[pattern.ToString()].SuccessfulPredictio

                        patternStats[pattern.ToString()].TotalReturn += bullishR
                        patternStats[pattern.ToString()].MaxReturn = Math.Max(pa
                        patternStats[pattern.ToString()].MinReturn = Math.Min(pa
                    }
                    else if (isBearish)
                    {
```

```
                        // For bearish patterns, success is defined as price mov
                        if (minPrice < entryPrice - 5 * TickSize)
                            patternStats[pattern.ToString()].SuccessfulPredictic

                        patternStats[pattern.ToString()].TotalReturn += bearishR
                        patternStats[pattern.ToString()].MaxReturn = Math.Max(pa
                        patternStats[pattern.ToString()].MinReturn = Math.Min(pa
                    }
                }
            }
        }

        // Class to store pattern statistics
        private class PatternStats
        {
            public int TotalOccurrences { get; set; }
            public int SuccessfulPredictions { get; set; }
            public double TotalReturn { get; set; }
            public double MaxReturn { get; set; }
            public double MinReturn { get; set; }

            public PatternStats()
            {
                TotalOccurrences = 0;
                SuccessfulPredictions = 0;
                TotalReturn = 0;
                MaxReturn = double.MinValue;
                MinReturn = double.MaxValue;
            }
        }
    }
}
```

Common Issues and Troubleshooting

This section addresses common issues encountered when working with candlestick patterns in NinjaTrader.

Pattern Detection Issues

Issue: Patterns are not being detected as expected.

Possible Causes: - Incorrect pattern parameters - Insufficient trend strength - Price scale affecting pattern recognition - Incorrect usage of the CandleStickPattern method

Solutions: - Verify the pattern parameters and trend strength values - Check the documentation for the specific pattern requirements - Ensure the price scale is appropriate for the instrument - Confirm the correct usage of the CandleStickPattern method

Example:

```
// Debugging pattern detection
protected override void OnBarUpdate()
{
    // Print candle properties for debugging
    Print("Bar " + CurrentBar + ": Open=" + Open[0] + ", High=" + High[0] + ", L

    // Test pattern detection with different trend strengths
    for (int strength = 1; strength <= 10; strength++)
    {
        if (CandleStickPattern(ChartPattern.BullishEngulfing, strength)[0] == 1)
        {
            Print("Bullish Engulfing detected with trend strength " + strength);
        }
    }
}
```

Performance Issues

Issue: Strategies based on candlestick patterns have poor performance.

Possible Causes: - Relying solely on candlestick patterns without confirmation - Using patterns in inappropriate market conditions - Improper risk management - Insufficient backtesting

Solutions: - Combine patterns with other technical indicators for confirmation - Use patterns in appropriate market contexts (trend, support/resistance, etc.) - Implement proper risk management (stop loss, position sizing, etc.) - Conduct thorough backtesting across different market conditions

Example:

```
// Improving pattern-based strategy performance
protected override void OnBarUpdate()
{
    // Wait for enough bars
    if (CurrentBar < 20)
        return;

    // Only trade during market hours
    if (!IsMarketHours())
        return;

    // Only trade in trending markets
    if (!IsTrending())
        return;

    // Only trade with confirmation
    if (!HasConfirmation())
```

```
        return;

    // Check for patterns
    if (CandleStickPattern(ChartPattern.BullishEngulfing, 4)[0] == 1)
    {
        // Enter with proper position sizing
        int quantity = CalculatePositionSize();
        EnterLong(quantity);

        // Set proper stop loss and profit target
        SetStopLoss(CalculationMode.Price, CalculateStopLossPrice());
        SetProfitTarget(CalculationMode.Price, CalculateProfitTargetPrice());
    }
}

private bool IsMarketHours()
{
    // Check if current time is within regular market hours
    return ToTime(Time[0]) >= 93000 && ToTime(Time[0]) <= 160000;
}

private bool IsTrending()
{
    // Check if market is trending using ADX
    return ADX(14)[0] > 25;
}

private bool HasConfirmation()
{
    // Check for confirmation using other indicators
    return RSI(14)[0] < 30 && MACD(12, 26, 9).Diff[0] > MACD(12, 26, 9).Diff[1];
}

private int CalculatePositionSize()
{
    // Calculate position size based on risk percentage
    double riskPercentage = 0.01; // 1% risk
    double accountValue = Account.Get(AccountItem.CashValue, Currency.UsDollar);
    double riskAmount = accountValue * riskPercentage;
    double stopLossAmount = Math.Abs(Close[0] - CalculateStopLossPrice());

    return Math.Max(1, (int)(riskAmount / stopLossAmount));
}

private double CalculateStopLossPrice()
{
    // Calculate stop loss price based on recent swing low
    return Math.Min(Low[0], Low[1]) - 2 * TickSize;
}

private double CalculateProfitTargetPrice()
```

```
{
    // Calculate profit target price based on risk-reward ratio
    double stopLossDistance = Math.Abs(Close[0] - CalculateStopLossPrice());
    return Close[0] + stopLossDistance * 2; // 1:2 risk-reward ratio
}
```

Custom Pattern Implementation Issues

Issue: Custom pattern detection is not working correctly.

Possible Causes: - Incorrect pattern definition - Logic errors in the detection code - Insufficient testing of edge cases - Improper handling of price data

Solutions: - Verify the pattern definition against standard references - Debug the detection code with Print statements - Test the detection code with known pattern examples - Ensure proper handling of price data (open, high, low, close)

Example:

```
// Debugging custom pattern detection
private bool IsBullishPinBar()
{
    // Calculate candle parts
    double body = Math.Abs(Close[0] - Open[0]);
    double upperWick = High[0] - Math.Max(Open[0], Close[0]);
    double lowerWick = Math.Min(Open[0], Close[0]) - Low[0];
    double totalRange = High[0] - Low[0];

    // Print values for debugging
    Print("Bar " + CurrentBar + " - Body: " + body + ", Upper Wick: " + upperWic

    // Pin Bar criteria
    bool hasLongLowerWick = lowerWick > body * 2 && lowerWick > upperWick * 3;
    bool hasSmallBody = body < totalRange * 0.3;
    bool isInDowntrend = Close[1] < Close[5] && Close[2] < Close[6];

    // Print criteria results
    Print("Has Long Lower Wick: " + hasLongLowerWick + ", Has Small Body: " + ha

    return hasLongLowerWick && hasSmallBody && isInDowntrend;
}
```

This comprehensive guide covers all aspects of candlestick patterns in NinjaTrader, providing traders with the knowledge and tools to effectively implement candlestick pattern-based trading strategies.

NinjaTrader Order Handling Systems

This comprehensive guide documents all aspects of order handling in NinjaTrader, including order types, order management, and execution.

Table of Contents

- Introduction to Order Handling
- Order Types
- Order Properties
- Order State Transitions
- Order Management
- Order Execution
- Order Placement Techniques
- Programmatic Order Handling
- Common Order Handling Patterns
- Troubleshooting Order Issues

Introduction to Order Handling

NinjaTrader provides a robust order handling system that allows traders to submit, modify, and cancel orders with precision. Understanding the order handling system is crucial for developing effective trading strategies and managing risk.

Order Flow in NinjaTrader

The typical order flow in NinjaTrader follows these steps: 1. Order creation (manually or programmatically) 2. Order submission to the broker/exchange 3. Order confirmation and state updates 4. Order execution (full or partial) 5. Position management 6. Order cancellation or modification (if needed)

Order Handling Approaches

NinjaTrader offers two primary approaches to order handling:

1. **Manual Order Handling**: Direct control over order submission, modification, and cancellation through UI or code.
2. **Managed Order Handling**: Simplified approach where NinjaTrader manages the complexities of order handling.

Order Types

NinjaTrader supports a wide range of order types to accommodate various trading strategies and market conditions.

Market Orders

Description: Executes immediately at the best available price.

Properties: - No price specification - Guaranteed execution (but not price) - Subject to slippage

Example Usage:

```
// Submitting a market order
EnterLong(1); // Managed approach
```

```
// Submitting a market order (unmanaged approach)
SubmitOrder(0, OrderAction.Buy, OrderType.Market, 1, 0, 0, "", "Market Entry");
```

Limit Orders

Description: Executes at the specified price or better.

Properties: - Price specification required - Not guaranteed to execute - Provides price improvement

Example Usage:

```
// Submitting a limit order
EnterLongLimit(1, Close[0] - 2 * TickSize); // Managed approach
```

```
// Submitting a limit order (unmanaged approach)
SubmitOrder(0, OrderAction.Buy, OrderType.Limit, 1, 0, Close[0] - 2 * TickSize,
```

Stop Orders

Description: Becomes a market order when the specified stop price is reached.

Properties: - Price specification required - Triggers at stop price - Executes as market order after triggering

Example Usage:

```
// Submitting a stop order
EnterLongStop(1, High[0] + 2 * TickSize); // Managed approach
```

```
// Submitting a stop order (unmanaged approach)
SubmitOrder(0, OrderAction.Buy, OrderType.Stop, 1, 0, High[0] + 2 * TickSize, ""
```

Stop-Limit Orders

Description: Combines stop and limit orders; becomes a limit order when the stop price is reached.

Properties: - Requires both stop price and limit price - Two-stage execution process - Provides price control after triggering

Example Usage:

```
// Submitting a stop-limit order
EnterLongStopLimit(1, High[0] + 2 * TickSize, High[0] + 4 * TickSize); // Manage
```

```
// Submitting a stop-limit order (unmanaged approach)
SubmitOrder(0, OrderAction.Buy, OrderType.StopLimit, 1, 0, High[0] + 2 * TickSiz
```

Market-If-Touched (MIT) Orders

Description: Becomes a market order when the specified price is reached.

Properties: - Similar to stop orders but in the opposite direction - Triggers when price touches the specified level - Executes as market order after triggering

Example Usage:

```
// MIT orders are implemented using limit orders in the managed approach
EnterLongLimit(1, Low[0] - 5 * TickSize); // Buy MIT
```

```
// MIT orders in unmanaged approach
SubmitOrder(0, OrderAction.Buy, OrderType.MIT, 1, 0, Low[0] - 5 * TickSize, "",
```

Simulated Stop Orders

Description: Stop orders simulated by NinjaTrader rather than the broker.

Properties: - Monitored by NinjaTrader - Converts to market order when triggered - Useful for markets that don't support native stop orders

Example Usage:

```
// Simulated stop orders are used automatically when needed
// Enable simulated order processing
Strategy.UnmanagedPosition = false;
Strategy.IncludeTradeHistoryInBacktest = true;
Strategy.OrderFillResolution = OrderFillResolution.Standard;
Strategy.SetOrderQuantity = SetOrderQuantity.Strategy;
Strategy.Slippage = 1;
```

```
Strategy.StartBehavior = StartBehavior.WaitUntilFlat;
Strategy.TimeInForce = TimeInForce.Day;
Strategy.TraceOrders = true;
Strategy.UseOnOrderUpdate = true;
Strategy.UseOnPositionUpdate = true;
```

Order Properties

Orders in NinjaTrader have various properties that define their behavior and characteristics.

Basic Order Properties

- **Action**: Buy or Sell
- **OrderType**: Market, Limit, Stop, StopLimit, etc.
- **Quantity**: Number of contracts/shares
- **Price**: Limit price (for limit orders)
- **StopPrice**: Trigger price (for stop orders)
- **TimeInForce**: Day, GTC (Good Till Cancelled), GTD (Good Till Date), etc.
- **OCO (One-Cancels-Other)**: Links orders so that when one is filled, the others are cancelled

Advanced Order Properties

- **Name**: Custom identifier for the order
- **OrderId**: Unique identifier assigned by NinjaTrader
- **OrderState**: Current state of the order (e.g., Working, Filled, Cancelled)
- **Filled**: Number of contracts/shares already filled
- **AverageFillPrice**: Average price of filled contracts/shares
- **ConnectionName**: The connection through which the order was placed
- **FromEntrySignal**: Indicates if the order was generated from an entry signal
- **IsSimulated**: Indicates if the order is simulated
- **OrderDescription**: Description of the order

Order State Transitions

Orders in NinjaTrader go through various states from creation to completion.

Primary Order States

- **Initialized**: Order has been created but not yet submitted
- **PendingSubmit**: Order has been submitted but not yet acknowledged by the broker
- **Working**: Order is active in the market
- **PendingChange**: Order change has been requested but not yet acknowledged
- **PendingCancel**: Order cancellation has been requested but not yet acknowledged
- **Cancelled**: Order has been cancelled
- **Rejected**: Order has been rejected by the broker

- **PartFilled**: Order has been partially filled
- **Filled**: Order has been completely filled

State Transition Flow

1. **Initialized → PendingSubmit**: Order is submitted
2. **PendingSubmit → Working**: Order is accepted by the broker
3. **Working → PartFilled**: Order is partially executed
4. **PartFilled → Filled**: Order is fully executed
5. **Working/PartFilled → PendingChange**: Order modification is requested
6. **PendingChange → Working**: Order modification is accepted
7. **Working/PartFilled → PendingCancel**: Order cancellation is requested
8. **PendingCancel → Cancelled**: Order is cancelled

Order Management

NinjaTrader provides various tools and techniques for managing orders throughout their lifecycle.

Order Submission

Orders can be submitted through: - Chart Trading - SuperDOM - Basic Entry - FX Board - FX Pro - Order Ticket - Programmatically via NinjaScript

Order Modification

Active orders can be modified to change: - Price - Quantity - Stop Price - Time in Force - OCO Group

Order Cancellation

Orders can be cancelled through: - Order display windows - Chart Trading - SuperDOM - Programmatically via NinjaScript

Order Grouping

NinjaTrader supports grouping orders in various ways: - **OCO (One-Cancels-Other)**: Links orders so that when one is filled, the others are cancelled - **Bracket Orders**: Entry order with attached stop loss and profit target - **ATM Strategies**: Advanced order management templates (covered in detail in the ATM Strategies section)

Order Execution

Understanding how orders are executed is crucial for developing effective trading strategies.

Execution Venues

Orders can be routed to different execution venues: - Exchanges - ECNs (Electronic Communication Networks) - Market Makers - Simulated Execution (for backtesting)

Execution Quality Factors

Several factors affect execution quality: - **Latency**: Time delay between order submission and execution - **Slippage**: Difference between expected execution price and actual execution price - **Partial Fills**: Orders that are filled in multiple parts at different prices - **Rejections**: Orders that are not accepted by the broker/exchange

Execution Reports

NinjaTrader provides detailed execution reports that include: - Fill price - Fill quantity - Fill time - Commission - Exchange fees - P&L (Profit and Loss)

Order Placement Techniques

NinjaTrader supports various order placement techniques to accommodate different trading styles.

Chart Trading

Description: Place orders directly on the chart.

Features: - Visual order placement - Drag-and-drop order modification - Quick access to common order types - Visual representation of working orders

Example Usage: 1. Right-click on the chart 2. Select "Chart Trading" to enable 3. Click on the chart to place orders 4. Drag orders to modify price

SuperDOM

Description: Dynamic Order Management interface for rapid order placement and modification.

Features: - Price ladder display - One-click order entry - Visual market depth - Quick order modification - ATM Strategy integration

Example Usage: 1. Open SuperDOM from the Control Center 2. Click on the price ladder to place orders 3. Right-click on orders to modify or cancel 4. Use hotkeys for rapid order placement

Basic Entry

Description: Simple interface for quick order entry.

Features: - Streamlined order entry - Quick access to common order types - ATM Strategy integration - Position management

Example Usage: 1. Open Basic Entry from the Control Center 2. Select instrument, quantity,

and order type 3. Click Buy or Sell to place the order

Order Ticket

Description: Detailed order entry form with all order parameters.

Features: - Access to all order properties - Advanced order types - Detailed order specifications - OCO grouping

Example Usage: 1. Open Order Ticket from the Control Center 2. Fill in all order details 3. Click Submit to place the order

Programmatic Order Handling

NinjaScript provides powerful capabilities for programmatic order handling in strategies and indicators.

Managed Approach

Description: Simplified order handling where NinjaTrader manages the complexities.

Features: - Automatic position tracking - Simplified order methods - Automatic order management - Strategy position management

Example Usage:

```
// Entry methods
EnterLong();
EnterShort();
EnterLongLimit(price);
EnterShortLimit(price);
EnterLongStop(stopPrice);
EnterShortStop(stopPrice);
EnterLongStopLimit(stopPrice, limitPrice);
EnterShortStopLimit(stopPrice, limitPrice);

// Exit methods
ExitLong();
ExitShort();
ExitLongLimit(price);
ExitShortLimit(price);
ExitLongStop(stopPrice);
ExitShortStop(stopPrice);
ExitLongStopLimit(stopPrice, limitPrice);
ExitShortStopLimit(stopPrice, limitPrice);
```

Unmanaged Approach

Description: Direct control over order submission, modification, and cancellation.

Features: - Complete control over order lifecycle - Advanced order handling capabilities - Custom order management logic - Direct access to order properties

Example Usage:

```
// Enable unmanaged order handling
protected override void OnStateChange()
{
    if (State == State.SetDefaults)
    {
        // Other properties...
        IsUnmanaged = true;
    }
}

// Submit an order
protected override void OnBarUpdate()
{
    if (CurrentBar < 20)
        return;

    // Submit a buy limit order
    string orderId = SubmitOrder(0, OrderAction.Buy, OrderType.Limit, 1, 0, Clos

    // Store the order ID for later reference
    entryOrderId = orderId;
}

// Handle order updates
protected override void OnOrderUpdate(Order order, double limitPrice, double sto
{
    if (order.Name == "Long Entry" && orderState == OrderState.Filled)
    {
        // Submit exit orders
        SubmitOrder(0, OrderAction.Sell, OrderType.Limit, 1, 0, averageFillPrice
        SubmitOrder(0, OrderAction.Sell, OrderType.Stop, 1, 0, averageFillPrice
    }
}
```

Order Submission Methods

Managed Approach: - `EnterLong()` : Enter a long position with a market order - `EnterShort()` : Enter a short position with a market order - `EnterLongLimit()` : Enter a long position with a limit order - `EnterShortLimit()` : Enter a short position with a limit order - `EnterLongStop()` : Enter a long position with a stop order - `EnterShortStop()` : Enter a short position with a stop order - `EnterLongStopLimit()` : Enter a long position with a stop-limit order - `EnterShortStopLimit()` : Enter a short position with a stop-limit order

Unmanaged Approach: - `SubmitOrder()` : Submit any type of order with full control over all parameters

Order Modification Methods

Managed Approach: - `ChangeOrder()` : Change an existing order's price or quantity

Unmanaged Approach: - `ChangeOrder()` : Change any property of an existing order

Order Cancellation Methods

Managed Approach: - `ExitLong()` : Exit a long position with a market order - `ExitShort()` : Exit a short position with a market order - `CancelOrder()` : Cancel a specific order

Unmanaged Approach: - `CancelOrder()` : Cancel a specific order - `CancelAllOrders()` : Cancel all orders for the strategy

Common Order Handling Patterns

These patterns demonstrate effective order handling techniques for common trading scenarios.

Entry with Stop Loss and Profit Target

```
// Managed approach
protected override void OnBarUpdate()
{
    if (CurrentBar < 20)
        return;

    // Entry condition
    if (CrossAbove(SMA(14), SMA(28), 1))
    {
        // Enter long with stop loss and profit target
        EnterLong("Entry", 1, stopPrice: Low[0] - 2 * TickSize, profitTarget: Hi
    }
}
```

```
// Unmanaged approach
private string entryOrderId = string.Empty;
private string stopOrderId = string.Empty;
private string targetOrderId = string.Empty;

protected override void OnBarUpdate()
{
    if (CurrentBar < 20 || IsIntrabarOrder)
        return;
```

```
    // Entry condition
    if (CrossAbove(SMA(14), SMA(28), 1) && entryOrderId == string.Empty && Posit
    {
        // Submit entry order
        entryOrderId = SubmitOrder(0, OrderAction.Buy, OrderType.Market, 1, 0, 0
    }
}

protected override void OnOrderUpdate(Order order, double limitPrice, double sto
{
    // Handle entry order fill
    if (order.Name == "Entry" && orderState == OrderState.Filled)
    {
        // Submit stop loss
        stopOrderId = SubmitOrder(0, OrderAction.Sell, OrderType.Stop, 1, 0, ave

        // Submit profit target
        targetOrderId = SubmitOrder(0, OrderAction.Sell, OrderType.Limit, 1, 0,

        // Link orders as OCO
        Order stopOrder = GetOrder(stopOrderId);
        Order targetOrder = GetOrder(targetOrderId);

        if (stopOrder != null && targetOrder != null)
            NinjaTrader.NinjaScript.OrderUtilities.SubmitOCO("ExitOCO", new[] {
    }

    // Reset order IDs when orders are filled or cancelled
    if (order.Name == "Entry" && (orderState == OrderState.Cancelled || orderSta
        entryOrderId = string.Empty;

    if ((order.Name == "StopLoss" || order.Name == "Target") && orderState == Or
    {
        stopOrderId = string.Empty;
        targetOrderId = string.Empty;
        entryOrderId = string.Empty;
    }
}
```

Scaling In and Out of Positions

```
// Managed approach for scaling in
protected override void OnBarUpdate()
{
    if (CurrentBar < 20)
        return;
```

```
    // Initial entry
    if (CrossAbove(RSI(14), 30, 1) && Position.MarketPosition == MarketPosition.
    {
        EnterLong(1, "Initial Entry");
    }

    // Scale in
    if (Position.MarketPosition == MarketPosition.Long && Close[0] > EntryPrice(
    {
        EnterLong(1, "Scale In");
    }

    // Scale out
    if (Position.MarketPosition == MarketPosition.Long && Position.Quantity > 0)
    {
        if (Close[0] >= EntryPrice() + 10 * TickSize)
        {
            ExitLong(Position.Quantity / 2, "Partial Exit", "");
        }

        if (Close[0] >= EntryPrice() + 20 * TickSize)
        {
            ExitLong("Final Exit", "");
        }
    }
}
```

```
// Unmanaged approach for scaling in and out
private List<double> entryPrices = new List<double>();
private int maxPositionSize = 3;

protected override void OnBarUpdate()
{
    if (CurrentBar < 20 || IsIntrabarOrder)
        return;

    // Initial entry
    if (CrossAbove(RSI(14), 30, 1) && Position.MarketPosition == MarketPosition.
    {
        SubmitOrder(0, OrderAction.Buy, OrderType.Market, 1, 0, 0, string.Empty,
    }

    // Scale in
    if (Position.MarketPosition == MarketPosition.Long && Close[0] > GetAverageE
    {
        SubmitOrder(0, OrderAction.Buy, OrderType.Market, 1, 0, 0, string.Empty,
    }

    // Scale out partial
```

```
    if (Position.MarketPosition == MarketPosition.Long && Position.Quantity > 1
    {
        SubmitOrder(0, OrderAction.Sell, OrderType.Market, Position.Quantity / 2
    }

    // Final exit
    if (Position.MarketPosition == MarketPosition.Long && Position.Quantity > 0
    {
        SubmitOrder(0, OrderAction.Sell, OrderType.Market, Position.Quantity, 0,
    }
}

protected override void OnOrderUpdate(Order order, double limitPrice, double stc
{
    if ((order.Name == "Initial Entry" || order.Name == "Scale In") && orderStat
    {
        entryPrices.Add(averageFillPrice);
    }

    if ((order.Name == "Partial Exit" || order.Name == "Final Exit") && orderSta
    {
        // Remove entry prices as we exit
        int filledQuantity = filled;
        while (filledQuantity > 0 && entryPrices.Count > 0)
        {
            entryPrices.RemoveAt(0);
            filledQuantity--;
        }
    }
}

private double GetAverageEntryPrice()
{
    if (entryPrices.Count == 0)
        return 0;

    double sum = 0;
    foreach (double price in entryPrices)
        sum += price;

    return sum / entryPrices.Count;
}
```

Breakout Entry with Trailing Stop

```
// Managed approach
protected override void OnBarUpdate()
{
```

```
    if (CurrentBar < 20)
        return;

    double highestHigh = MAX(High, 20)[1];
    double lowestLow = MIN(Low, 20)[1];

    // Breakout entry
    if (Close[0] > highestHigh && Position.MarketPosition == MarketPosition.Flat
    {
        EnterLong("Breakout Entry");
    }

    // Trailing stop
    if (Position.MarketPosition == MarketPosition.Long)
    {
        double stopPrice = Math.Max(EntryPrice() - 10 * TickSize, Close[0] - 5 *
        ExitLongStop(stopPrice, "Trailing Stop", "");
    }
}
```

```
// Unmanaged approach
private string entryOrderId = string.Empty;
private string stopOrderId = string.Empty;
private double highestClose = 0;

protected override void OnBarUpdate()
{
    if (CurrentBar < 20 || IsIntrabarOrder)
        return;

    double highestHigh = MAX(High, 20)[1];

    // Breakout entry
    if (Close[0] > highestHigh && Position.MarketPosition == MarketPosition.Flat
    {
        entryOrderId = SubmitOrder(0, OrderAction.Buy, OrderType.Market, 1, 0, 0
    }

    // Update trailing stop
    if (Position.MarketPosition == MarketPosition.Long)
    {
        // Track highest close since entry
        highestClose = Math.Max(highestClose, Close[0]);

        // Calculate new stop price
        double stopPrice = Math.Max(Position.AveragePrice - 10 * TickSize, highe

        // Update or create stop order
        if (stopOrderId == string.Empty)
```

```
        {
            stopOrderId = SubmitOrder(0, OrderAction.Sell, OrderType.Stop, Posit
        }
        else
        {
            Order stopOrder = GetOrder(stopOrderId);
            if (stopOrder != null && stopOrder.StopPrice < stopPrice)
            {
                ChangeOrder(stopOrder, stopOrder.Quantity, stopPrice, stopOrder.
            }
        }
    }
}

protected override void OnOrderUpdate(Order order, double limitPrice, double stc
{
    if (order.Name == "Breakout Entry" && orderState == OrderState.Filled)
    {
        highestClose = Close[0];
    }

    if (order.Name == "Breakout Entry" && (orderState == OrderState.Cancelled ||
    {
        entryOrderId = string.Empty;
    }

    if (order.Name == "Trailing Stop" && orderState == OrderState.Filled)
    {
        stopOrderId = string.Empty;
        entryOrderId = string.Empty;
        highestClose = 0;
    }
}
```

Troubleshooting Order Issues

Common order-related issues and their solutions.

Order Rejection

Common Causes: - Insufficient margin/buying power - Invalid order parameters - Market closed or halted - Connection issues - Regulatory restrictions

Solutions: - Check account balance and margin requirements - Verify order parameters (price, quantity, etc.) - Ensure market is open and trading - Check connection status - Review regulatory restrictions for the instrument

Partial Fills

Common Causes: - Insufficient liquidity - Large order size - Fast-moving markets

Solutions: - Break large orders into smaller pieces - Use limit orders instead of market orders - Implement a scaling strategy - Consider time-slicing orders

Slippage

Common Causes: - Market volatility - Low liquidity - Fast-moving markets - News events

Solutions: - Use limit orders when possible - Avoid trading during high volatility periods - Consider liquidity when selecting instruments - Be cautious around news events

Order Stuck in Pending State

Common Causes: - Connection issues - Broker system problems - NinjaTrader internal issues

Solutions: - Check connection status - Restart NinjaTrader - Contact broker support - Use the Cancel All Orders function - Implement timeout handling in strategies

Order Execution Delays

Common Causes: - Network latency - Broker processing time - Market congestion - System load

Solutions: - Optimize network connection - Consider a different broker or connection - Avoid trading during extremely busy periods - Implement timeout handling in strategies

This comprehensive guide covers all aspects of order handling in NinjaTrader. For information on Advanced Trade Management (ATM) strategies, exit strategies, and stop types, please refer to the dedicated sections in this knowledge base.

NinjaTrader ATM and Exit Strategies

This comprehensive guide documents all aspects of Advanced Trade Management (ATM) strategies and exit strategies in NinjaTrader.

Table of Contents

- Introduction to ATM Strategies
- ATM Strategy Components
- Creating and Managing ATM Strategies
- ATM Strategy Templates
- Exit Strategies
- ATM Strategy Methods in NinjaScript
- Advanced ATM Features
- Common ATM Strategy Patterns
- Troubleshooting ATM Strategies

Introduction to ATM Strategies

Advanced Trade Management (ATM) Strategies in NinjaTrader are pre-defined sets of orders and rules that automate trade management after entry. They provide a systematic approach to managing positions with predefined profit targets, stop losses, and other exit conditions.

What is an ATM Strategy?

An ATM Strategy is a collection of orders that represent entries, exits, stops, and targets, along with sub-strategies (Auto Breakeven, Auto Trail, etc.) that govern how these orders are managed. ATM Strategies are designed to provide discretionary traders with semi-automated features to manage their positions.

Benefits of ATM Strategies

- **Consistency**: Apply the same exit rules to every trade
- **Efficiency**: Automatically place stop loss and profit target orders
- **Flexibility**: Modify parameters on-the-fly during a trade
- **Discipline**: Enforce predefined risk management rules
- **Focus**: Concentrate on entry signals rather than exit management
- **Customization**: Create templates for different market conditions

ATM Strategy vs. NinjaScript Strategy

It's important to understand the difference between ATM Strategies and NinjaScript Strategies:

- **ATM Strategies**: Semi-automated trade management tools for discretionary traders
- **NinjaScript Strategies**: Fully automated trading systems that can generate both entry and exit signals

ATM Strategies can be used independently or in conjunction with NinjaScript Strategies. When used with NinjaScript Strategies, the NinjaScript Strategy generates entry signals, and the ATM Strategy manages the exits.

ATM Strategy Components

ATM Strategies consist of several components that work together to manage trades.

Stop Loss Orders

Stop loss orders are designed to limit risk by exiting a position when the market moves against you by a specified amount.

Types of Stop Loss Orders in ATM Strategies: - **Fixed Stop**: Set at a specific price level - **Calculated Stop**: Based on a formula (e.g., ATR multiple) - **Breakeven Stop**: Moves to entry price after a specified profit is reached - **Trailing Stop**: Follows price movement to lock in profits

Profit Target Orders

Profit target orders are designed to secure profits by exiting a position when a specified profit level is reached.

Types of Profit Target Orders in ATM Strategies: - **Fixed Target**: Set at a specific price level - **Multiple Targets**: Multiple orders at different price levels for scaling out - **Calculated Target**: Based on a formula (e.g., risk-reward ratio)

Auto Breakeven

Auto Breakeven automatically moves the stop loss order to the entry price (plus/minus offset) once the position has reached a specified profit amount.

Parameters: - **Profit Trigger**: Amount of profit required to activate Auto Breakeven - **Plus/Minus Ticks**: Offset from the entry price (can be positive or negative)

Auto Trail

Auto Trail automatically adjusts the stop loss order as the market moves in your favor, maintaining a specified distance from the current price.

Parameters: - **Profit Trigger**: Amount of profit required to activate Auto Trail - **Trail Amount**: Distance to maintain between current price and stop loss - **Trail Style**: Method used to calculate the trailing stop (e.g., fixed amount, percentage, ATR)

Quantity Management

ATM Strategies allow for sophisticated quantity management, including:

- **Multiple Targets**: Distribute position quantity across multiple profit targets
- **Scale Out**: Reduce position size at predetermined levels
- **Partial Exits**: Exit portions of a position at different times

Creating and Managing ATM Strategies

ATM Strategies can be created and managed through various NinjaTrader interfaces.

Creating ATM Strategies

ATM Strategies can be created through: - SuperDOM - Chart Trader - Basic Entry - FX Pro - Order Ticket

Basic Steps to Create an ATM Strategy: 1. Select the instrument and quantity 2. Define stop loss and profit target parameters 3. Configure Auto Breakeven and Auto Trail settings (if desired) 4. Save the strategy as a template (optional) 5. Submit the entry order

Example: Creating an ATM Strategy in SuperDOM

1. Open a SuperDOM for your desired instrument
2. Set the quantity in the Qty field
3. Right-click on the price ladder where you want to place your entry order
4. Select Buy or Sell and the desired order type
5. In the ATM Strategy Selection Mode dropdown, select "Create new strategy"
6. Configure stop loss and profit target parameters
7. Click "Submit" to place the order and activate the ATM Strategy

Managing Active ATM Strategies

Once an ATM Strategy is active, you can manage it through: - SuperDOM - Chart Trader - Position Display - Strategy Display

Common Management Actions: - Modify stop loss or profit target prices - Enable/disable Auto Breakeven or Auto Trail - Add additional targets - Close the entire position - Deactivate the ATM Strategy

Example: Modifying an Active ATM Strategy in SuperDOM

1. Locate the active ATM Strategy in the SuperDOM
2. Drag the stop loss or profit target lines to new price levels
3. Right-click on the stop loss or profit target and select "Properties" to modify parameters
4. Use the ATM Strategy control panel to enable/disable Auto Breakeven or Auto Trail

ATM Strategy Templates

ATM Strategy Templates allow you to save and reuse ATM Strategy configurations.

Creating ATM Strategy Templates

1. Configure an ATM Strategy with your desired parameters
2. In the ATM Strategy Selection Mode dropdown, select "Save strategy as template"
3. Enter a name for the template
4. Click "Save"

Managing ATM Strategy Templates

ATM Strategy Templates can be managed through: - Control Center → Tools → Options → ATM Strategy - ATM Strategy Selection Mode dropdown in order entry interfaces

Template Management Actions: - Rename templates - Delete templates - Set default templates - Import/export templates

Example: Using ATM Strategy Templates for Different Market Conditions

```
// Template for trending markets
```

```
Name: Trend_Following
Stop Loss: 15 ticks
Profit Target: 30 ticks
Auto Breakeven: Enabled (Profit Trigger: 10 ticks, Plus/Minus: 1 tick)
Auto Trail: Enabled (Profit Trigger: 15 ticks, Trail Amount: 10 ticks)

// Template for volatile markets
Name: Volatile_Market
Stop Loss: 25 ticks
Profit Target 1: 15 ticks (50% of position)
Profit Target 2: 30 ticks (50% of position)
Auto Breakeven: Disabled
Auto Trail: Enabled (Profit Trigger: 10 ticks, Trail Amount: 15 ticks)

// Template for range-bound markets
Name: Range_Bound
Stop Loss: 10 ticks
Profit Target: 15 ticks
Auto Breakeven: Enabled (Profit Trigger: 7 ticks, Plus/Minus: 0 ticks)
Auto Trail: Disabled
```

Exit Strategies

Exit strategies define how and when to exit a position. NinjaTrader provides various exit strategies that can be used independently or as part of ATM Strategies.

Types of Exit Strategies

Stop Loss Strategies

- **Fixed Price Stop**: Exit at a specific price level
- **Tick-Based Stop**: Exit after a specified number of ticks against the position
- **Percentage-Based Stop**: Exit after a specified percentage move against the position
- **ATR-Based Stop**: Exit based on Average True Range (covered in detail in the ATR Trailing Stops section)
- **Time-Based Stop**: Exit after a specified time period
- **Indicator-Based Stop**: Exit based on indicator signals

Profit Target Strategies

- **Fixed Price Target**: Exit at a specific price level
- **Tick-Based Target**: Exit after a specified number of ticks in favor of the position
- **Percentage-Based Target**: Exit after a specified percentage move in favor of the position
- **Risk-Reward Target**: Exit at a multiple of the risk taken
- **Indicator-Based Target**: Exit based on indicator signals

Breakeven Strategies

- **Simple Breakeven**: Move stop to entry price
- **Breakeven Plus**: Move stop to entry price plus an offset
- **Partial Position Breakeven**: Move stop to breakeven after partial profit is taken

Trailing Stop Strategies

- **Fixed Tick Trail**: Maintain a fixed distance from the highest/lowest price
- **Percentage Trail**: Maintain a percentage distance from the highest/lowest price
- **ATR Trail**: Maintain a distance based on Average True Range
- **Chandelier Exit**: Trail based on ATR from highest high (for longs) or lowest low (for shorts)
- **Parabolic SAR Trail**: Use Parabolic SAR indicator for trailing

Scale-Out Strategies

- **Equal Parts**: Exit equal portions at multiple levels
- **Decreasing Size**: Exit larger portions early, smaller portions later
- **Increasing Size**: Exit smaller portions early, larger portions later

Combination Strategies

- **Bracket Orders**: Combination of stop loss and profit target
- **OCO (One-Cancels-Other)**: Multiple exit orders where filling one cancels the others
- **Multi-Stage Exit**: Different exit strategies at different stages of the trade

Implementing Exit Strategies

Exit strategies can be implemented through: - ATM Strategies - Manual order placement - NinjaScript programming

Example: Multi-Stage Exit Strategy Implementation

```
// Multi-stage exit strategy in NinjaScript
private bool stageOneExitPlaced = false;
private bool stageTwoExitPlaced = false;

protected override void OnBarUpdate()
{
    // Entry logic
    if (CrossAbove(SMA(14), SMA(28), 1) && Position.MarketPosition == MarketPosi
    {
        EnterLong();
    }

    // Exit logic - Stage One (move to breakeven after 10 ticks profit)
    if (Position.MarketPosition == MarketPosition.Long && !stageOneExitPlaced)
    {
        if (Close[0] >= Position.AveragePrice + 10 * TickSize)
        {
            ExitLongStop(Position.AveragePrice, "Breakeven Stop");
```

```
            stageOneExitPlaced = true;
        }
    }

    // Exit logic - Stage Two (trail stop after 20 ticks profit)
    if (Position.MarketPosition == MarketPosition.Long && stageOneExitPlaced &&
    {
        if (Close[0] >= Position.AveragePrice + 20 * TickSize)
        {
            double trailPrice = Close[0] - 10 * TickSize;
            ExitLongStop(trailPrice, "Trailing Stop");
            stageTwoExitPlaced = true;
        }
    }

    // Exit logic - Stage Three (update trailing stop)
    if (Position.MarketPosition == MarketPosition.Long && stageTwoExitPlaced)
    {
        double newTrailPrice = Close[0] - 10 * TickSize;
        ExitLongStop(Math.Max(GetExitOrder("Trailing Stop").StopPrice, newTrailP
    }
}
```

ATM Strategy Methods in NinjaScript

NinjaTrader provides a set of methods for working with ATM Strategies in NinjaScript.

ATM Strategy Management Methods

- **AtmStrategyCreate()**: Creates a new ATM Strategy
- **AtmStrategyClose()**: Closes an ATM Strategy
- **AtmStrategyChangeEntryOrder()**: Changes the parameters of an entry order
- **AtmStrategyChangeStopTarget()**: Changes the parameters of a stop loss or profit target order
- **AtmStrategyCancelEntryOrder()**: Cancels an entry order

ATM Strategy Monitoring Methods

- **GetAtmStrategyEntryOrderStatus()**: Gets the status of an entry order
- **GetAtmStrategyMarketPosition()**: Gets the market position of an ATM Strategy
- **GetAtmStrategyPositionAveragePrice()**: Gets the average entry price of an ATM Strategy
- **GetAtmStrategyPositionQuantity()**: Gets the position quantity of an ATM Strategy
- **GetAtmStrategyRealizedProfitLoss()**: Gets the realized profit/loss of an ATM Strategy
- **GetAtmStrategyStopTargetOrderStatus()**: Gets the status of a stop loss or profit target order
- **GetAtmStrategyUniqueId()**: Gets the unique identifier of an ATM Strategy

- **GetAtmStrategyUnrealizedProfitLoss()**: Gets the unrealized profit/loss of an ATM Strategy

Example: Using ATM Strategy Methods in NinjaScript

```
private string atmStrategyId = string.Empty;
private string entryOrderId = string.Empty;

protected override void OnBarUpdate()
{
    if (CurrentBar < 20)
        return;

    // Entry condition
    if (CrossAbove(SMA(14), SMA(28), 1) && atmStrategyId == string.Empty)
    {
        // Create ATM Strategy with 1 contract, 10 tick stop loss, 20 tick profi
        atmStrategyId = AtmStrategyCreate(OrderAction.Buy, OrderType.Market, 0,
            "MyATMStrategy", "Entry",
            new[] { 10 * TickSize }, // Stop loss offset
            new[] { 20 * TickSize }, // Profit target offset
            new[] { 1 }, // Quantity for each target
            new[] { false }, // Is stop loss percentage
            new[] { false }); // Is profit target percentage

        Print("Created ATM Strategy: " + atmStrategyId);
    }

    // Monitor ATM Strategy
    if (atmStrategyId != string.Empty)
    {
        // Get market position
        MarketPosition position = GetAtmStrategyMarketPosition(atmStrategyId);

        // Get average price
        double avgPrice = GetAtmStrategyPositionAveragePrice(atmStrategyId);

        // Get unrealized P&L
        double unrealizedPL = GetAtmStrategyUnrealizedProfitLoss(atmStrategyId);

        Print("ATM Strategy: " + atmStrategyId + ", Position: " + position +
              ", Avg Price: " + avgPrice + ", Unrealized P&L: " + unrealizedPL);

        // Check if position is flat (ATM Strategy is complete)
        if (position == MarketPosition.Flat && GetAtmStrategyPositionQuantity(at
        {
            double realizedPL = GetAtmStrategyRealizedProfitLoss(atmStrategyId);
            Print("ATM Strategy complete. Realized P&L: " + realizedPL);
            atmStrategyId = string.Empty;
        }
```

```
    }
}
```

Advanced ATM Features

NinjaTrader provides several advanced features for ATM Strategies.

Shadow Strategy

Shadow Strategy allows you to simulate the execution of an ATM Strategy without actually placing orders. This is useful for testing and learning.

How to Use Shadow Strategy: 1. Enable Shadow Strategy in the ATM Strategy control panel 2. Place an entry order as usual 3. The ATM Strategy will simulate the execution of the strategy without placing actual orders

Auto Chase

Auto Chase automatically adjusts the price of a limit entry order if the market moves away from the order price.

Parameters: - **Chase Ticks**: Number of ticks to adjust the order price - **Max Chase Ticks**: Maximum number of ticks to chase the market - **Chase Interval**: Time interval between price adjustments

Auto Reverse

Auto Reverse automatically reverses your position when an ATM Strategy is closed by a stop loss order.

How to Use Auto Reverse: 1. Enable Auto Reverse in the ATM Strategy control panel 2. When a stop loss order is filled, a new ATM Strategy is automatically created in the opposite direction

Close at Time

Close at Time automatically closes an ATM Strategy at a specified time.

Parameters: - **Close Time**: Time at which to close the ATM Strategy - **Action**: Action to take (e.g., market order, limit order)

Indicator Tracking

Indicator Tracking allows you to link an ATM Strategy to an indicator for dynamic stop loss and profit target placement.

How to Use Indicator Tracking: 1. Enable Indicator Tracking in the ATM Strategy control panel 2. Select the indicator to track 3. Configure the tracking parameters 4. The ATM Strategy will

adjust stop loss and profit target orders based on the indicator values

Common ATM Strategy Patterns

These patterns demonstrate effective ATM Strategy configurations for common trading scenarios.

Trend Following ATM Strategy

Configuration: - **Stop Loss**: 15 ticks or 1.5 * ATR - **Profit Target**: 30 ticks or 3 * ATR - **Auto Breakeven**: Enabled (Profit Trigger: 10 ticks, Plus/Minus: 1 tick) - **Auto Trail**: Enabled (Profit Trigger: 15 ticks, Trail Amount: 10 ticks)

Implementation:

```
// Create Trend Following ATM Strategy
string atmStrategyId = AtmStrategyCreate(OrderAction.Buy, OrderType.Market, 0, 0
    "TrendFollowing", "Entry",
    new[] { 15 * TickSize }, // Stop loss offset
    new[] { 30 * TickSize }, // Profit target offset
    new[] { 1 }, // Quantity for each target
    new[] { false }, // Is stop loss percentage
    new[] { false }); // Is profit target percentage

// Enable Auto Breakeven
// Note: This would typically be done through the UI, but can be simulated in cc
```

Scalping ATM Strategy

Configuration: - **Stop Loss**: 5 ticks - **Profit Target**: 8 ticks - **Auto Breakeven**: Enabled (Profit Trigger: 4 ticks, Plus/Minus: 0 ticks) - **Auto Trail**: Disabled

Implementation:

```
// Create Scalping ATM Strategy
string atmStrategyId = AtmStrategyCreate(OrderAction.Buy, OrderType.Market, 0, 0
    "Scalping", "Entry",
    new[] { 5 * TickSize }, // Stop loss offset
    new[] { 8 * TickSize }, // Profit target offset
    new[] { 1 }, // Quantity for each target
    new[] { false }, // Is stop loss percentage
    new[] { false }); // Is profit target percentage
```

Multi-Target ATM Strategy

Configuration: - **Stop Loss**: 15 ticks - **Profit Target 1**: 10 ticks (50% of position) - **Profit**

Target 2: 20 ticks (30% of position) - **Profit Target 3**: 30 ticks (20% of position) - **Auto Breakeven**: Enabled (Profit Trigger: 10 ticks, Plus/Minus: 0 ticks) - **Auto Trail**: Enabled for remaining position after first target is hit (Trail Amount: 10 ticks)

Implementation:

```
// Create Multi-Target ATM Strategy
string atmStrategyId = AtmStrategyCreate(OrderAction.Buy, OrderType.Market, 0, 0
    "MultiTarget", "Entry",
    new[] { 15 * TickSize, 15 * TickSize, 15 * TickSize }, // Stop loss offset f
    new[] { 10 * TickSize, 20 * TickSize, 30 * TickSize }, // Profit target offs
    new[] { 5, 3, 2 }, // Quantity for each target (total: 10 contracts)
    new[] { false, false, false }, // Is stop loss percentage
    new[] { false, false, false }); // Is profit target percentage
```

Breakout ATM Strategy

Configuration: - **Stop Loss**: 12 ticks - **Profit Target**: 25 ticks - **Auto Breakeven**: Enabled (Profit Trigger: 15 ticks, Plus/Minus: 2 ticks) - **Auto Trail**: Enabled (Profit Trigger: 20 ticks, Trail Amount: 15 ticks)

Implementation:

```
// Create Breakout ATM Strategy
string atmStrategyId = AtmStrategyCreate(OrderAction.Buy, OrderType.Stop, High[1
    "Breakout", "Entry",
    new[] { 12 * TickSize }, // Stop loss offset
    new[] { 25 * TickSize }, // Profit target offset
    new[] { 1 }, // Quantity for each target
    new[] { false }, // Is stop loss percentage
    new[] { false }); // Is profit target percentage
```

Gap Fill ATM Strategy

Configuration: - **Stop Loss**: 8 ticks - **Profit Target**: Distance to gap fill level - **Auto Breakeven**: Enabled (Profit Trigger: 10 ticks, Plus/Minus: 1 tick) - **Auto Trail**: Disabled

Implementation:

```
// Assuming we've identified a gap and calculated the gap fill level
double gapFillLevel = CalculateGapFillLevel();
double entryPrice = Close[0];
double profitTargetOffset = Math.Abs(gapFillLevel - entryPrice);

// Create Gap Fill ATM Strategy
string atmStrategyId = AtmStrategyCreate(
    gapFillLevel < entryPrice ? OrderAction.Sell : OrderAction.Buy,
```

```
    OrderType.Market, 0, 0, TimeInForce.Day, 1,
    "GapFill", "Entry",
    new[] { 8 * TickSize }, // Stop loss offset
    new[] { profitTargetOffset }, // Profit target offset
    new[] { 1 }, // Quantity for each target
    new[] { false }, // Is stop loss percentage
    new[] { false }); // Is profit target percentage
```

Troubleshooting ATM Strategies

Common ATM Strategy issues and their solutions.

ATM Strategy Not Created

Common Causes: - Insufficient account balance - Invalid parameters - Connection issues

Solutions: - Check account balance and margin requirements - Verify all parameters are valid - Check connection status - Restart NinjaTrader if necessary

Stop Loss or Profit Target Not Working

Common Causes: - Incorrect price levels - Market gapping through levels - Order rejection by broker

Solutions: - Verify price levels are appropriate for market conditions - Use market orders for stop loss in volatile conditions - Check for order rejection messages - Consider using simulated stops for illiquid markets

Auto Breakeven Not Triggering

Common Causes: - Profit trigger not reached - Incorrect configuration - Market moving too quickly

Solutions: - Verify profit trigger level - Check Auto Breakeven settings - Consider lowering profit trigger in volatile markets

Auto Trail Not Working

Common Causes: - Profit trigger not reached - Incorrect configuration - Market moving too quickly

Solutions: - Verify profit trigger level - Check Auto Trail settings - Consider using a larger trail amount in volatile markets

ATM Strategy Deactivated Unexpectedly

Common Causes: - All orders filled or cancelled - Connection loss - System restart

Solutions: - Check order status - Verify connection stability - Implement recovery procedures for unexpected deactivation

This comprehensive guide covers all aspects of ATM Strategies and exit strategies in NinjaTrader. For information on specific stop types, including ATR trailing stops, please refer to the dedicated section in this knowledge base.

NinjaTrader Stop Types and ATR Trailing Stops

This comprehensive guide documents all stop types available in NinjaTrader, with special focus on ATR trailing stops and their implementation.

Table of Contents

- Introduction to Stop Orders
- Basic Stop Types
- Advanced Stop Types
- ATR Trailing Stops
- Implementing Stop Types in NinjaScript
- Stop Order Placement Techniques
- Common Stop Strategies
- Troubleshooting Stop Orders

Introduction to Stop Orders

Stop orders are essential risk management tools that automatically exit positions when certain conditions are met. They help traders limit losses and protect profits by providing automated exit mechanisms.

Purpose of Stop Orders

- **Risk Management**: Limit potential losses on trades
- **Profit Protection**: Lock in profits as trades move in your favor
- **Emotion Removal**: Automate exits to remove emotional decision-making
- **Trade Management**: Free up attention for new opportunities

Stop Order Mechanics

In NinjaTrader, stop orders work through these mechanisms:

1. **Order Submission**: Stop order is submitted to the broker or simulated locally
2. **Monitoring**: Price is continuously monitored against stop conditions
3. **Triggering**: When stop conditions are met, the stop order is triggered
4. **Execution**: The triggered stop order becomes a market or limit order for execution

Stop Order Considerations

- **Slippage**: Difference between stop trigger price and actual execution price
- **Liquidity**: Market depth affects execution quality
- **Volatility**: High volatility can lead to larger slippage
- **Visibility**: Stop orders visible to the market may be targeted
- **Simulated vs. Server-Side**: Where the stop is monitored and triggered

Basic Stop Types

NinjaTrader supports several basic stop types for different trading scenarios.

Standard Stop Loss

Description: Exits a position when price reaches a specified level against the trade direction.

Parameters: - **Stop Price**: Price level at which the stop is triggered - **Order Type**: Market or Limit (after triggering)

Example Usage:

```
// Setting a standard stop loss in a managed strategy
protected override void OnBarUpdate()
{
    if (Position.MarketPosition == MarketPosition.Long)
    {
        // Set stop loss 10 ticks below entry price
        SetStopLoss(CalculationMode.Ticks, 10);
    }
}
```

```
// Setting a standard stop loss in an unmanaged strategy
protected override void OnBarUpdate()
{
    if (Position.MarketPosition == MarketPosition.Long)
    {
        // Submit stop order 10 ticks below entry price
        SubmitOrder(0, OrderAction.Sell, OrderType.Stop, Position.Quantity, 0,
            Position.AveragePrice - 10 * TickSize, string.Empty, "Stop Loss");
    }
}
```

Breakeven Stop

Description: Moves the stop loss to the entry price (or slightly better/worse) after the position reaches a specified profit.

Parameters: - **Profit Trigger**: Amount of profit required to activate the breakeven stop - **Plus/Minus Ticks**: Offset from the entry price (can be positive or negative)

Example Usage:

```
// Implementing a breakeven stop in a managed strategy
private bool breakevenEnabled = false;
private double profitTrigger = 10 * TickSize;
private double breakevenOffset = 2 * TickSize; // 2 ticks above entry for long p

protected override void OnBarUpdate()
{
    if (Position.MarketPosition == MarketPosition.Long && !breakevenEnabled)
    {
        // Check if profit trigger is reached
        if (Close[0] >= Position.AveragePrice + profitTrigger)
        {
            // Set stop to breakeven plus offset
            SetStopLoss(CalculationMode.Price, Position.AveragePrice + breakeven
            breakevenEnabled = true;
        }
    }
}
```

```
// Implementing a breakeven stop in an unmanaged strategy
private string stopOrderId = string.Empty;
private bool breakevenEnabled = false;
private double profitTrigger = 10 * TickSize;
private double breakevenOffset = 2 * TickSize; // 2 ticks above entry for long p

protected override void OnBarUpdate()
{
    if (Position.MarketPosition == MarketPosition.Long && !breakevenEnabled)
    {
        // Check if profit trigger is reached
        if (Close[0] >= Position.AveragePrice + profitTrigger)
        {
            // Cancel existing stop order
            if (!string.IsNullOrEmpty(stopOrderId))
            {
                CancelOrder(stopOrderId);
            }

            // Submit new stop order at breakeven plus offset
            stopOrderId = SubmitOrder(0, OrderAction.Sell, OrderType.Stop, Posit
                Position.AveragePrice + breakevenOffset, string.Empty, "Breakeve

            breakevenEnabled = true;
        }
    }
}
```

Fixed Trailing Stop

Description: Maintains a fixed distance from the highest/lowest price reached since entry.

Parameters: - **Trail Amount**: Fixed distance to maintain from highest/lowest price - **Step Size**: Minimum price movement required to update the stop (optional)

Example Usage:

```
// Setting a fixed trailing stop in a managed strategy
protected override void OnBarUpdate()
{
    if (Position.MarketPosition == MarketPosition.Long)
    {
        // Set trailing stop 10 ticks below current price
        SetTrailStop(CalculationMode.Ticks, 10);
    }
}
```

```
// Implementing a fixed trailing stop in an unmanaged strategy
private string stopOrderId = string.Empty;
private double highestPrice = 0;
private double trailAmount = 10 * TickSize;

protected override void OnBarUpdate()
{
    if (Position.MarketPosition == MarketPosition.Long)
    {
        // Update highest price
        highestPrice = Math.Max(highestPrice, High[0]);

        // Calculate new stop price
        double newStopPrice = highestPrice - trailAmount;

        // Update stop order if needed
        if (string.IsNullOrEmpty(stopOrderId))
        {
            // Create initial stop order
            stopOrderId = SubmitOrder(0, OrderAction.Sell, OrderType.Stop, Posit
                newStopPrice, string.Empty, "Trailing Stop");
        }
        else
        {
            // Get current stop order
            Order stopOrder = GetOrder(stopOrderId);

            // Update stop price if new price is higher
            if (stopOrder != null && newStopPrice > stopOrder.StopPrice)
```

```
            {
                ChangeOrder(stopOrder, stopOrder.Quantity, newStopPrice, stopOrd
            }
        }
    }
}
```

Percentage Trailing Stop

Description: Maintains a percentage distance from the highest/lowest price reached since entry.

Parameters: - **Percentage**: Percentage distance to maintain from highest/lowest price - **Step Size**: Minimum price movement required to update the stop (optional)

Example Usage:

```
// Implementing a percentage trailing stop
private string stopOrderId = string.Empty;
private double highestPrice = 0;
private double trailPercentage = 1.0; // 1% trail

protected override void OnBarUpdate()
{
    if (Position.MarketPosition == MarketPosition.Long)
    {
        // Update highest price
        highestPrice = Math.Max(highestPrice, High[0]);

        // Calculate new stop price (1% below highest price)
        double newStopPrice = highestPrice * (1 - trailPercentage / 100);

        // Update stop order if needed
        if (string.IsNullOrEmpty(stopOrderId))
        {
            // Create initial stop order
            stopOrderId = SubmitOrder(0, OrderAction.Sell, OrderType.Stop, Posit
                newStopPrice, string.Empty, "Percentage Trailing Stop");
        }
        else
        {
            // Get current stop order
            Order stopOrder = GetOrder(stopOrderId);

            // Update stop price if new price is higher
            if (stopOrder != null && newStopPrice > stopOrder.StopPrice)
            {
                ChangeOrder(stopOrder, stopOrder.Quantity, newStopPrice, stopOrd
            }
```

```
        }
    }
}
```

Time-Based Stop

Description: Exits a position after a specified time period.

Parameters: - **Time Period**: Duration after which to exit the position - **Order Type**: Market or Limit (for exit)

Example Usage:

```
// Implementing a time-based stop
private DateTime entryTime;
private bool timeStopEnabled = false;
private TimeSpan timeLimit = new TimeSpan(0, 30, 0); // 30 minutes

protected override void OnBarUpdate()
{
    // Record entry time
    if (Position.MarketPosition == MarketPosition.Flat && entryTime != DateTime.
    {
        entryTime = DateTime.MinValue;
        timeStopEnabled = false;
    }
    else if (Position.MarketPosition != MarketPosition.Flat && entryTime == Date
    {
        entryTime = Time[0];
        timeStopEnabled = true;
    }

    // Check if time limit is reached
    if (timeStopEnabled && Time[0] >= entryTime.Add(timeLimit))
    {
        if (Position.MarketPosition == MarketPosition.Long)
            ExitLong();
        else if (Position.MarketPosition == MarketPosition.Short)
            ExitShort();

        timeStopEnabled = false;
    }
}
```

Indicator-Based Stop

Description: Exits a position based on indicator signals.

Parameters: - **Indicator**: Technical indicator to monitor - **Threshold**: Value or condition that triggers the stop - **Order Type**: Market or Limit (for exit)

Example Usage:

```
// Implementing a stop based on RSI crossing below 30
private RSI rsi;

protected override void OnStateChange()
{
    if (State == State.Configure)
    {
        rsi = RSI(14);
    }
}

protected override void OnBarUpdate()
{
    if (Position.MarketPosition == MarketPosition.Long)
    {
        // Exit if RSI crosses below 30
        if (CrossBelow(rsi, 30, 1))
        {
            ExitLong();
        }
    }
}
```

Advanced Stop Types

NinjaTrader supports several advanced stop types for sophisticated risk management.

Chandelier Exit

Description: A trailing stop based on ATR, measured from the highest high (for longs) or lowest low (for shorts) since entry.

Parameters: - **ATR Period**: Period for ATR calculation - **ATR Multiple**: Multiplier for ATR value - **Lookback Period**: Period for highest high/lowest low calculation

Example Usage:

```
// Implementing a Chandelier Exit
private ATR atr;
private double highestHigh = 0;
private string stopOrderId = string.Empty;
private int atrPeriod = 14;
private double atrMultiple = 3.0;
```

```
protected override void OnStateChange()
{
    if (State == State.Configure)
    {
        atr = ATR(atrPeriod);
    }
}

protected override void OnBarUpdate()
{
    if (Position.MarketPosition == MarketPosition.Long)
    {
        // Update highest high
        highestHigh = Math.Max(highestHigh, High[0]);

        // Calculate Chandelier Exit level
        double chandelierLevel = highestHigh - (atr[0] * atrMultiple);

        // Update stop order
        if (string.IsNullOrEmpty(stopOrderId))
        {
            stopOrderId = SubmitOrder(0, OrderAction.Sell, OrderType.Stop, Posit
                chandelierLevel, string.Empty, "Chandelier Exit");
        }
        else
        {
            Order stopOrder = GetOrder(stopOrderId);
            if (stopOrder != null && chandelierLevel > stopOrder.StopPrice)
            {
                ChangeOrder(stopOrder, stopOrder.Quantity, chandelierLevel, stop
            }
        }
    }
}
```

Parabolic SAR Stop

Description: Uses the Parabolic SAR indicator as a trailing stop.

Parameters: - **Acceleration Factor**: Starting acceleration factor - **Acceleration Limit**: Maximum acceleration factor - **Order Type**: Market or Limit (for exit)

Example Usage:

```
// Implementing a Parabolic SAR stop
private ParabolicSAR psar;
private string stopOrderId = string.Empty;
```

```
protected override void OnStateChange()
{
    if (State == State.Configure)
    {
        psar = ParabolicSAR(0.02, 0.2);
    }
}

protected override void OnBarUpdate()
{
    if (Position.MarketPosition == MarketPosition.Long)
    {
        // Use Parabolic SAR as stop level
        double stopLevel = psar[0];

        // Update stop order
        if (string.IsNullOrEmpty(stopOrderId))
        {
            stopOrderId = SubmitOrder(0, OrderAction.Sell, OrderType.Stop, Posit
                stopLevel, string.Empty, "PSAR Stop");
        }
        else
        {
            Order stopOrder = GetOrder(stopOrderId);
            if (stopOrder != null)
            {
                ChangeOrder(stopOrder, stopOrder.Quantity, stopLevel, stopOrder.
            }
        }
    }
}
```

Volatility-Based Stop

Description: Adjusts stop distance based on market volatility.

Parameters: - **Volatility Measure**: Indicator used to measure volatility (e.g., ATR, Standard Deviation) - **Volatility Multiple**: Multiplier for volatility measure - **Minimum Distance**: Minimum stop distance regardless of volatility

Example Usage:

```
// Implementing a volatility-based stop using Standard Deviation
private StdDev stdDev;
private string stopOrderId = string.Empty;
private double volatilityMultiple = 2.0;
private double minimumDistance = 5 * TickSize;

protected override void OnStateChange()
```

```
{
    if (State == State.Configure)
    {
        stdDev = StdDev(14, 1);
    }
}

protected override void OnBarUpdate()
{
    if (Position.MarketPosition == MarketPosition.Long)
    {
        // Calculate stop distance based on volatility
        double stopDistance = Math.Max(stdDev[0] * volatilityMultiple, minimumDi

        // Calculate stop level
        double stopLevel = Position.AveragePrice - stopDistance;

        // Update stop order
        if (string.IsNullOrEmpty(stopOrderId))
        {
            stopOrderId = SubmitOrder(0, OrderAction.Sell, OrderType.Stop, Posit
                stopLevel, string.Empty, "Volatility Stop");
        }
        else
        {
            Order stopOrder = GetOrder(stopOrderId);
            if (stopOrder != null)
            {
                ChangeOrder(stopOrder, stopOrder.Quantity, stopLevel, stopOrder.
            }
        }
    }
}
```

Ratcheting Stop

Description: A trailing stop that only moves at predetermined intervals.

Parameters: - **Initial Stop Distance**: Initial distance from entry price - **Ratchet Amount**: Amount to move the stop at each interval - **Profit Interval**: Profit amount required to move the stop

Example Usage:

```
// Implementing a ratcheting stop
private string stopOrderId = string.Empty;
private double initialStopDistance = 15 * TickSize;
private double ratchetAmount = 5 * TickSize;
private double profitInterval = 10 * TickSize;
```

```
private int ratchetLevel = 0;

protected override void OnBarUpdate()
{
    if (Position.MarketPosition == MarketPosition.Long)
    {
        // Calculate current profit
        double currentProfit = Close[0] - Position.AveragePrice;

        // Calculate ratchet level based on profit
        int newRatchetLevel = (int)(currentProfit / profitInterval);

        // Update stop if ratchet level increased
        if (newRatchetLevel > ratchetLevel)
        {
            ratchetLevel = newRatchetLevel;

            // Calculate new stop level
            double stopLevel = Position.AveragePrice - initialStopDistance + (ra

            // Update stop order
            if (string.IsNullOrEmpty(stopOrderId))
            {
                stopOrderId = SubmitOrder(0, OrderAction.Sell, OrderType.Stop, P
                    stopLevel, string.Empty, "Ratchet Stop");
            }
            else
            {
                Order stopOrder = GetOrder(stopOrderId);
                if (stopOrder != null)
                {
                    ChangeOrder(stopOrder, stopOrder.Quantity, stopLevel, stopOr
                }
            }
        }
    }
}
```

Moving Average Stop

Description: Uses a moving average as a trailing stop.

Parameters: - **MA Type**: Type of moving average (SMA, EMA, etc.) - **MA Period**: Period for moving average calculation - **Offset**: Additional distance from the moving average

Example Usage:

```
// Implementing a moving average stop
private EMA ema;
```

```
private string stopOrderId = string.Empty;
private double offset = 2 * TickSize;

protected override void OnStateChange()
{
    if (State == State.Configure)
    {
        ema = EMA(20);
    }
}

protected override void OnBarUpdate()
{
    if (Position.MarketPosition == MarketPosition.Long)
    {
        // Calculate stop level based on EMA
        double stopLevel = ema[0] - offset;

        // Update stop order
        if (string.IsNullOrEmpty(stopOrderId))
        {
            stopOrderId = SubmitOrder(0, OrderAction.Sell, OrderType.Stop, Posit
                stopLevel, string.Empty, "MA Stop");
        }
        else
        {
            Order stopOrder = GetOrder(stopOrderId);
            if (stopOrder != null && stopLevel > stopOrder.StopPrice)
            {
                ChangeOrder(stopOrder, stopOrder.Quantity, stopLevel, stopOrder.
            }
        }
    }
}
```

ATR Trailing Stops

ATR (Average True Range) trailing stops are among the most effective stop types because they adapt to market volatility. This section provides detailed information on implementing ATR trailing stops in NinjaTrader.

Basic ATR Trailing Stop

Description: A trailing stop that maintains a distance of X times the ATR from the highest/lowest price.

Parameters: - **ATR Period**: Period for ATR calculation (typically 14) - **ATR Multiple**: Multiplier for ATR value (typically 2-3) - **Calculation Frequency**: How often to recalculate the stop (e.g.,

every bar, every tick)

Example Usage:

```
// Implementing a basic ATR trailing stop
private ATR atr;
private string stopOrderId = string.Empty;
private double highestPrice = 0;
private int atrPeriod = 14;
private double atrMultiple = 2.5;

protected override void OnStateChange()
{
    if (State == State.Configure)
    {
        atr = ATR(atrPeriod);
    }
}

protected override void OnBarUpdate()
{
    if (Position.MarketPosition == MarketPosition.Long)
    {
        // Update highest price
        highestPrice = Math.Max(highestPrice, High[0]);

        // Calculate ATR trailing stop level
        double stopLevel = highestPrice - (atr[0] * atrMultiple);

        // Update stop order
        if (string.IsNullOrEmpty(stopOrderId))
        {
            stopOrderId = SubmitOrder(0, OrderAction.Sell, OrderType.Stop, Posit
                stopLevel, string.Empty, "ATR Trailing Stop");
        }
        else
        {
            Order stopOrder = GetOrder(stopOrderId);
            if (stopOrder != null && stopLevel > stopOrder.StopPrice)
            {
                ChangeOrder(stopOrder, stopOrder.Quantity, stopLevel, stopOrder.
            }
        }
    }
    else if (Position.MarketPosition == MarketPosition.Flat)
    {
        // Reset tracking variables when flat
        highestPrice = 0;
        stopOrderId = string.Empty;
    }
```

```
}
```

ATR Trailing Stop with Activation Threshold

Description: An ATR trailing stop that only activates after the position reaches a specified profit.

Parameters: - **ATR Period**: Period for ATR calculation - **ATR Multiple**: Multiplier for ATR value - **Profit Trigger**: Amount of profit required to activate the trailing stop - **Initial Stop**: Stop level before the trailing stop activates

Example Usage:

```
// Implementing an ATR trailing stop with activation threshold
private ATR atr;
private string stopOrderId = string.Empty;
private double highestPrice = 0;
private bool trailActive = false;
private int atrPeriod = 14;
private double atrMultiple = 2.5;
private double profitTrigger = 10 * TickSize;
private double initialStopDistance = 15 * TickSize;

protected override void OnStateChange()
{
    if (State == State.Configure)
    {
        atr = ATR(atrPeriod);
    }
}

protected override void OnBarUpdate()
{
    if (Position.MarketPosition == MarketPosition.Long)
    {
        // Update highest price
        highestPrice = Math.Max(highestPrice, High[0]);

        // Check if trailing stop should be activated
        if (!trailActive && (highestPrice - Position.AveragePrice) >= profitTrig
        {
            trailActive = true;
        }

        // Calculate stop level
        double stopLevel;
        if (trailActive)
        {
            // Use ATR trailing stop
```

```
            stopLevel = highestPrice - (atr[0] * atrMultiple);
        }
        else
        {
            // Use initial fixed stop
            stopLevel = Position.AveragePrice - initialStopDistance;
        }

        // Update stop order
        if (string.IsNullOrEmpty(stopOrderId))
        {
            stopOrderId = SubmitOrder(0, OrderAction.Sell, OrderType.Stop, Posit
                stopLevel, string.Empty, "ATR Trailing Stop");
        }
        else
        {
            Order stopOrder = GetOrder(stopOrderId);
            if (stopOrder != null && stopLevel > stopOrder.StopPrice)
            {
                ChangeOrder(stopOrder, stopOrder.Quantity, stopLevel, stopOrder.
            }
        }
    }
    else if (Position.MarketPosition == MarketPosition.Flat)
    {
        // Reset tracking variables when flat
        highestPrice = 0;
        stopOrderId = string.Empty;
        trailActive = false;
    }
}
```

Stepped ATR Trailing Stop

Description: An ATR trailing stop that moves in steps rather than continuously.

Parameters: - **ATR Period**: Period for ATR calculation - **ATR Multiple**: Multiplier for ATR value - **Step Size**: Minimum price movement required to update the stop

Example Usage:

```
// Implementing a stepped ATR trailing stop
private ATR atr;
private string stopOrderId = string.Empty;
private double highestPrice = 0;
private double lastStopLevel = 0;
private int atrPeriod = 14;
private double atrMultiple = 2.5;
private double stepSize = 5 * TickSize;
```

```
protected override void OnStateChange()
{
    if (State == State.Configure)
    {
        atr = ATR(atrPeriod);
    }
}

protected override void OnBarUpdate()
{
    if (Position.MarketPosition == MarketPosition.Long)
    {
        // Update highest price
        highestPrice = Math.Max(highestPrice, High[0]);

        // Calculate ATR trailing stop level
        double currentStopLevel = highestPrice - (atr[0] * atrMultiple);

        // Only update if step size is reached
        if (lastStopLevel == 0 || currentStopLevel - lastStopLevel >= stepSize)
        {
            lastStopLevel = currentStopLevel;

            // Update stop order
            if (string.IsNullOrEmpty(stopOrderId))
            {
                stopOrderId = SubmitOrder(0, OrderAction.Sell, OrderType.Stop, P
                    lastStopLevel, string.Empty, "Stepped ATR Trailing Stop");
            }
            else
            {
                Order stopOrder = GetOrder(stopOrderId);
                if (stopOrder != null)
                {
                    ChangeOrder(stopOrder, stopOrder.Quantity, lastStopLevel, st
                }
            }
        }
    }
    else if (Position.MarketPosition == MarketPosition.Flat)
    {
        // Reset tracking variables when flat
        highestPrice = 0;
        stopOrderId = string.Empty;
        lastStopLevel = 0;
    }
}
```

Multi-Timeframe ATR Trailing Stop

Description: An ATR trailing stop that uses a higher timeframe for ATR calculation.

Parameters: - **ATR Period**: Period for ATR calculation - **ATR Multiple**: Multiplier for ATR value - **Timeframe**: Higher timeframe for ATR calculation

Example Usage:

```
// Implementing a multi-timeframe ATR trailing stop
private ATR atr;
private string stopOrderId = string.Empty;
private double highestPrice = 0;
private int atrPeriod = 14;
private double atrMultiple = 2.5;

protected override void OnStateChange()
{
    if (State == State.Configure)
    {
        // Use daily timeframe for ATR calculation
        atr = ATR(atrPeriod, MarketDataType.Daily);
    }
}

protected override void OnBarUpdate()
{
    if (Position.MarketPosition == MarketPosition.Long)
    {
        // Update highest price
        highestPrice = Math.Max(highestPrice, High[0]);

        // Calculate ATR trailing stop level using daily ATR
        double stopLevel = highestPrice - (atr[0] * atrMultiple);

        // Update stop order
        if (string.IsNullOrEmpty(stopOrderId))
        {
            stopOrderId = SubmitOrder(0, OrderAction.Sell, OrderType.Stop, Posit
                stopLevel, string.Empty, "Multi-TF ATR Trailing Stop");
        }
        else
        {
            Order stopOrder = GetOrder(stopOrderId);
            if (stopOrder != null && stopLevel > stopOrder.StopPrice)
            {
                ChangeOrder(stopOrder, stopOrder.Quantity, stopLevel, stopOrder.
            }
        }
    }
```

```
    else if (Position.MarketPosition == MarketPosition.Flat)
    {
        // Reset tracking variables when flat
        highestPrice = 0;
        stopOrderId = string.Empty;
    }
}
```

ATR Trailing Stop with Lookback Period

Description: An ATR trailing stop that uses the highest high/lowest low over a specified lookback period.

Parameters: - **ATR Period**: Period for ATR calculation - **ATR Multiple**: Multiplier for ATR value - **Lookback Period**: Period for highest high/lowest low calculation

Example Usage:

```
// Implementing an ATR trailing stop with lookback period
private ATR atr;
private string stopOrderId = string.Empty;
private int atrPeriod = 14;
private double atrMultiple = 2.5;
private int lookbackPeriod = 5;

protected override void OnBarUpdate()
{
    if (Position.MarketPosition == MarketPosition.Long)
    {
        // Find highest high over lookback period
        double highestHigh = High[0];
        for (int i = 1; i < lookbackPeriod; i++)
        {
            highestHigh = Math.Max(highestHigh, High[i]);
        }

        // Calculate ATR trailing stop level
        double stopLevel = highestHigh - (atr[0] * atrMultiple);

        // Update stop order
        if (string.IsNullOrEmpty(stopOrderId))
        {
            stopOrderId = SubmitOrder(0, OrderAction.Sell, OrderType.Stop, Posit
                stopLevel, string.Empty, "Lookback ATR Trailing Stop");
        }
        else
        {
            Order stopOrder = GetOrder(stopOrderId);
            if (stopOrder != null && stopLevel > stopOrder.StopPrice)
```

```
            {
                ChangeOrder(stopOrder, stopOrder.Quantity, stopLevel, stopOrder.
            }
        }
    }
    else if (Position.MarketPosition == MarketPosition.Flat)
    {
        // Reset tracking variables when flat
        stopOrderId = string.Empty;
    }
}
```

Complete ATR Trailing Stop Implementation

This comprehensive implementation combines multiple features for a robust ATR trailing stop system.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.ComponentModel.DataAnnotations;
using NinjaTrader.Cbi;
using NinjaTrader.Data;
using NinjaTrader.NinjaScript;
using NinjaTrader.Core.FloatingPoint;
using NinjaTrader.NinjaScript.Indicators;

namespace NinjaTrader.NinjaScript.Strategies
{
    public class ATRTrailingStopStrategy : Strategy
    {
        private ATR atr;
        private string stopOrderId = string.Empty;
        private double highestPrice = 0;
        private double lowestPrice = double.MaxValue;
        private bool trailActive = false;

        [NinjaScriptProperty]
        [Range(1, int.MaxValue)]
        [Display(Name = "ATR Period", Description = "Period for ATR calculation"
        public int AtrPeriod { get; set; }

        [NinjaScriptProperty]
        [Range(0.1, double.MaxValue)]
        [Display(Name = "ATR Multiple", Description = "Multiplier for ATR value"
        public double AtrMultiple { get; set; }

        [NinjaScriptProperty]
        [Range(0, double.MaxValue)]
```

```
        [Display(Name = "Profit Trigger (Ticks)", Description = "Profit required
        public double ProfitTriggerTicks { get; set; }

        [NinjaScriptProperty]
        [Range(0, double.MaxValue)]
        [Display(Name = "Initial Stop (Ticks)", Description = "Initial stop dist
        public double InitialStopTicks { get; set; }

        [NinjaScriptProperty]
        [Display(Name = "Use Higher Timeframe", Description = "Use daily timefra
        public bool UseHigherTimeframe { get; set; }

        protected override void OnStateChange()
        {
            if (State == State.SetDefaults)
            {
                Description = "ATR Trailing Stop Strategy";
                Name = "ATRTrailingStop";

                // Default parameter values
                AtrPeriod = 14;
                AtrMultiple = 2.5;
                ProfitTriggerTicks = 10;
                InitialStopTicks = 15;
                UseHigherTimeframe = false;

                // Strategy settings
                IsUnmanaged = true;
                IncludeTradeHistoryInBacktest = true;
                IsAutoScale = true;
                IsFillLimitOnTouch = false;
            }
            else if (State == State.Configure)
            {
                // Add ATR indicator
                if (UseHigherTimeframe)
                    atr = ATR(AtrPeriod, MarketDataType.Daily);
                else
                    atr = ATR(AtrPeriod);

                // Add a simple entry condition for demonstration
                // In a real strategy, you would replace this with your own entr
                Add(new Plot(new Pen(System.Windows.Media.Colors.Blue, 2), PlotS
            }
            else if (State == State.Terminated)
            {
                // Clean up resources
            }
        }

        protected override void OnBarUpdate()
```

```
        {
            // Wait for enough bars to calculate indicators
            if (CurrentBar < 20)
                return;

            // Simple entry logic for demonstration
            // In a real strategy, you would replace this with your own entry lo
            if (CrossAbove(SMA(14), SMA(28), 1) && Position.MarketPosition == Ma
            {
                EnterLong();
            }

            // ATR Trailing Stop logic
            ManageTrailingStop();
        }

        private void EnterLong()
        {
            // Submit entry order
            string entryOrderId = SubmitOrder(0, OrderAction.Buy, OrderType.Mark

            // Reset tracking variables
            highestPrice = 0;
            trailActive = false;
        }

        private void ManageTrailingStop()
        {
            // Only manage stops when in a position
            if (Position.MarketPosition == MarketPosition.Long)
            {
                // Update highest price
                highestPrice = Math.Max(highestPrice, High[0]);

                // Check if trailing stop should be activated
                double profitTrigger = ProfitTriggerTicks * TickSize;
                if (!trailActive && (highestPrice - Position.AveragePrice) >= pr
                {
                    trailActive = true;
                }

                // Calculate stop level
                double stopLevel;
                if (trailActive)
                {
                    // Use ATR trailing stop
                    stopLevel = highestPrice - (atr[0] * AtrMultiple);
                }
                else
                {
                    // Use initial fixed stop
```

```
                    stopLevel = Position.AveragePrice - (InitialStopTicks * Tick
                }

                // Update stop order
                if (string.IsNullOrEmpty(stopOrderId))
                {
                    stopOrderId = SubmitOrder(0, OrderAction.Sell, OrderType.Sto
                        stopLevel, string.Empty, "ATR Trailing Stop");
                }
                else
                {
                    Order stopOrder = GetOrder(stopOrderId);
                    if (stopOrder != null && stopLevel > stopOrder.StopPrice)
                    {
                        ChangeOrder(stopOrder, stopOrder.Quantity, stopLevel, st
                    }
                }

                // Draw stop level on chart
                Draw.Line(this, "StopLine", false, 1, stopLevel, 0, stopLevel, S
            }
            else if (Position.MarketPosition == MarketPosition.Short)
            {
                // Update lowest price
                lowestPrice = Math.Min(lowestPrice, Low[0]);

                // Check if trailing stop should be activated
                double profitTrigger = ProfitTriggerTicks * TickSize;
                if (!trailActive && (Position.AveragePrice - lowestPrice) >= pro
                {
                    trailActive = true;
                }

                // Calculate stop level
                double stopLevel;
                if (trailActive)
                {
                    // Use ATR trailing stop
                    stopLevel = lowestPrice + (atr[0] * AtrMultiple);
                }
                else
                {
                    // Use initial fixed stop
                    stopLevel = Position.AveragePrice + (InitialStopTicks * Tick
                }

                // Update stop order
                if (string.IsNullOrEmpty(stopOrderId))
                {
                    stopOrderId = SubmitOrder(0, OrderAction.Buy, OrderType.Stop
                        stopLevel, string.Empty, "ATR Trailing Stop");
```

```
                }
                else
                {
                    Order stopOrder = GetOrder(stopOrderId);
                    if (stopOrder != null && stopLevel < stopOrder.StopPrice)
                    {
                        ChangeOrder(stopOrder, stopOrder.Quantity, stopLevel, st
                    }
                }

                // Draw stop level on chart
                Draw.Line(this, "StopLine", false, 1, stopLevel, 0, stopLevel, S
            }
            else if (Position.MarketPosition == MarketPosition.Flat)
            {
                // Reset tracking variables when flat
                highestPrice = 0;
                lowestPrice = double.MaxValue;
                stopOrderId = string.Empty;
                trailActive = false;

                // Remove stop line from chart
                RemoveDrawObject("StopLine");
            }
        }

        protected override void OnOrderUpdate(Order order, double limitPrice, do
        {
            // Track stop order ID
            if (order.Name == "ATR Trailing Stop" && orderState == OrderState.Ac
            {
                stopOrderId = order.OrderId;
            }

            // Reset stop order ID when filled or cancelled
            if (order.Name == "ATR Trailing Stop" && (orderState == OrderState.F
            {
                stopOrderId = string.Empty;
            }
        }

        protected override void OnExecutionUpdate(Execution execution, string ex
        {
            // Handle executions if needed
        }
    }
}
```

Implementing Stop Types in NinjaScript

This section provides guidance on implementing various stop types in NinjaScript strategies.

Managed vs. Unmanaged Approach

NinjaTrader offers two approaches to implementing stops in NinjaScript:

Managed Approach

Description: NinjaTrader manages the complexities of order handling.

Advantages: - Simpler implementation - Automatic position tracking - Built-in stop management functions

Example:

```
// Setting stops using managed approach
protected override void OnBarUpdate()
{
    if (CrossAbove(SMA(14), SMA(28), 1))
    {
        EnterLong();

        // Set stop loss and profit target
        SetStopLoss(CalculationMode.Ticks, 10);
        SetProfitTarget(CalculationMode.Ticks, 20);
    }
}
```

Unmanaged Approach

Description: Direct control over order submission, modification, and cancellation.

Advantages: - Complete control over order lifecycle - Advanced order handling capabilities - Custom stop management logic

Example:

```
// Setting stops using unmanaged approach
private string entryOrderId = string.Empty;
private string stopOrderId = string.Empty;
private string targetOrderId = string.Empty;

protected override void OnBarUpdate()
{
    if (CrossAbove(SMA(14), SMA(28), 1) && Position.MarketPosition == MarketPosi
    {
        // Submit entry order
        entryOrderId = SubmitOrder(0, OrderAction.Buy, OrderType.Market, 1, 0, 0
```

```
    }
}

protected override void OnOrderUpdate(Order order, double limitPrice, double stc
{
    // Handle entry order fill
    if (order.Name == "Entry" && orderState == OrderState.Filled)
    {
        // Submit stop loss order
        stopOrderId = SubmitOrder(0, OrderAction.Sell, OrderType.Stop, 1, 0,
            averageFillPrice - 10 * TickSize, string.Empty, "Stop Loss");

        // Submit profit target order
        targetOrderId = SubmitOrder(0, OrderAction.Sell, OrderType.Limit, 1, 0,
            averageFillPrice + 20 * TickSize, string.Empty, "Profit Target");

        // Link orders as OCO
        Order stopOrder = GetOrder(stopOrderId);
        Order targetOrder = GetOrder(targetOrderId);

        if (stopOrder != null && targetOrder != null)
            NinjaTrader.NinjaScript.OrderUtilities.SubmitOCO("ExitOCO", new[] {
    }
}
```

Stop Management Methods

NinjaTrader provides several methods for managing stops in NinjaScript strategies.

Managed Approach Methods

- **SetStopLoss()**: Sets a stop loss for the current position
- **SetTrailStop()**: Sets a trailing stop for the current position
- **SetProfitTarget()**: Sets a profit target for the current position

Example:

```
// Using managed approach methods
protected override void OnBarUpdate()
{
    if (CrossAbove(SMA(14), SMA(28), 1))
    {
        EnterLong();

        // Set fixed stop loss
        SetStopLoss(CalculationMode.Ticks, 10);

        // Set trailing stop
        SetTrailStop(CalculationMode.Ticks, 15);
```

```
        // Set profit target
        SetProfitTarget(CalculationMode.Ticks, 20);
    }
}
```

Unmanaged Approach Methods

- **SubmitOrder()**: Submits a new order
- **ChangeOrder()**: Changes an existing order
- **CancelOrder()**: Cancels an existing order

Example:

```
// Using unmanaged approach methods
private string stopOrderId = string.Empty;

protected override void OnBarUpdate()
{
    if (Position.MarketPosition == MarketPosition.Long)
    {
        // Calculate new stop price
        double newStopPrice = Low[0] - 5 * TickSize;

        // Update or create stop order
        if (string.IsNullOrEmpty(stopOrderId))
        {
            stopOrderId = SubmitOrder(0, OrderAction.Sell, OrderType.Stop, Posit
                newStopPrice, string.Empty, "Stop Loss");
        }
        else
        {
            Order stopOrder = GetOrder(stopOrderId);
            if (stopOrder != null && newStopPrice > stopOrder.StopPrice)
            {
                ChangeOrder(stopOrder, stopOrder.Quantity, newStopPrice, stopOrd
            }
        }
    }
}
```

Stop Order Event Handling

Proper event handling is crucial for managing stop orders effectively.

OnOrderUpdate

Description: Called when an order's state changes.

Usage: - Track order IDs - Handle order state transitions - Implement order management logic

Example:

```
protected override void OnOrderUpdate(Order order, double limitPrice, double stc
{
    // Track stop order ID
    if (order.Name == "Stop Loss" && orderState == OrderState.Accepted)
    {
        stopOrderId = order.OrderId;
    }

    // Reset stop order ID when filled or cancelled
    if (order.Name == "Stop Loss" && (orderState == OrderState.Filled || orderSt
    {
        stopOrderId = string.Empty;
    }

    // Handle order rejection
    if (orderState == OrderState.Rejected)
    {
        Print("Order rejected: " + order.Name + ", Error: " + error.ToString());
    }
}
```

OnExecutionUpdate

Description: Called when an order is executed (filled).

Usage: - Track executions - Calculate average entry/exit prices - Implement post-execution logic

Example:

```
protected override void OnExecutionUpdate(Execution execution, string executionI
{
    // Handle stop loss execution
    if (GetOrder(orderId).Name == "Stop Loss")
    {
        Print("Stop loss executed at " + price);

        // Reset tracking variables
        highestPrice = 0;
        lowestPrice = double.MaxValue;
    }
}
```

Stop Order Placement Techniques

This section covers various techniques for placing stop orders in NinjaTrader.

Fixed Price Stops

Description: Stop orders placed at specific price levels.

Example:

```
// Placing a fixed price stop
protected override void OnBarUpdate()
{
    if (Position.MarketPosition == MarketPosition.Long)
    {
        // Place stop at a specific support level
        double supportLevel = 1234.50;
        SetStopLoss(CalculationMode.Price, supportLevel);
    }
}
```

Indicator-Based Stops

Description: Stop orders placed based on indicator values.

Example:

```
// Placing a stop based on Bollinger Bands
private Bollinger bollinger;

protected override void OnStateChange()
{
    if (State == State.Configure)
    {
        bollinger = Bollinger(20, 2);
    }
}

protected override void OnBarUpdate()
{
    if (Position.MarketPosition == MarketPosition.Long)
    {
        // Place stop at lower Bollinger Band
        SetStopLoss(CalculationMode.Price, bollinger.Lower[0]);
    }
}
```

Chart Pattern Stops

Description: Stop orders placed based on chart patterns.

Example:

```
// Placing a stop below a swing low
protected override void OnBarUpdate()
{
    if (Position.MarketPosition == MarketPosition.Long)
    {
        // Find recent swing low
        double swingLow = Low[0];
        for (int i = 1; i < 10; i++)
        {
            if (Low[i] < Low[i-1] && Low[i] < Low[i+1])
            {
                swingLow = Low[i];
                break;
            }
        }

        // Place stop below swing low
        SetStopLoss(CalculationMode.Price, swingLow - 2 * TickSize);
    }
}
```

Time-Based Stops

Description: Stop orders that exit positions after a specified time.

Example:

```
// Implementing a time-based stop
private DateTime entryTime;

protected override void OnBarUpdate()
{
    // Record entry time
    if (Position.MarketPosition == MarketPosition.Flat && entryTime != DateTime.
    {
        entryTime = DateTime.MinValue;
    }
    else if (Position.MarketPosition != MarketPosition.Flat && entryTime == Date
    {
        entryTime = Time[0];
    }

    // Exit after 30 minutes
    if (Position.MarketPosition != MarketPosition.Flat && entryTime != DateTime.
    {
        TimeSpan timeInTrade = Time[0] - entryTime;
        if (timeInTrade.TotalMinutes >= 30)
```

```
        {
            if (Position.MarketPosition == MarketPosition.Long)
                ExitLong();
            else if (Position.MarketPosition == MarketPosition.Short)
                ExitShort();
        }
    }
}
```

Common Stop Strategies

These strategies demonstrate effective stop management techniques for common trading scenarios.

Breakout Trading with ATR Stops

Description: Enter on breakouts and use ATR-based stops.

Example:

```
private ATR atr;

protected override void OnStateChange()
{
    if (State == State.Configure)
    {
        atr = ATR(14);
    }
}

protected override void OnBarUpdate()
{
    // Wait for enough bars
    if (CurrentBar < 20)
        return;

    // Calculate 20-day high
    double highestHigh = MAX(High, 20)[1];

    // Breakout entry
    if (Close[0] > highestHigh && Position.MarketPosition == MarketPosition.Flat
    {
        EnterLong();

        // Set ATR-based stop
        SetStopLoss(CalculationMode.Price, Close[0] - (atr[0] * 2));
    }
```

```
    // Update trailing stop
    if (Position.MarketPosition == MarketPosition.Long)
    {
        double newStopPrice = Close[0] - (atr[0] * 2);
        SetStopLoss(CalculationMode.Price, newStopPrice);
    }
}
```

Trend Following with Chandelier Exit

Description: Enter on trend signals and use Chandelier Exit for stops.

Example:

```
private ATR atr;
private double highestHigh = 0;

protected override void OnStateChange()
{
    if (State == State.Configure)
    {
        atr = ATR(14);
    }
}

protected override void OnBarUpdate()
{
    // Wait for enough bars
    if (CurrentBar < 20)
        return;

    // Trend following entry
    if (CrossAbove(EMA(20), EMA(50), 1) && Position.MarketPosition == MarketPosi
    {
        EnterLong();
        highestHigh = High[0];
    }

    // Update highest high
    if (Position.MarketPosition == MarketPosition.Long)
    {
        highestHigh = Math.Max(highestHigh, High[0]);

        // Chandelier Exit
        double stopPrice = highestHigh - (atr[0] * 3);
        SetStopLoss(CalculationMode.Price, stopPrice);
    }
    else
    {
```

```
        highestHigh = 0;
    }
}
```

Range Trading with Volatility Stops

Description: Enter on range bounces and use volatility-based stops.

Example:

```
private ATR atr;

protected override void OnStateChange()
{
    if (State == State.Configure)
    {
        atr = ATR(14);
    }
}

protected override void OnBarUpdate()
{
    // Wait for enough bars
    if (CurrentBar < 50)
        return;

    // Calculate range
    double rangeHigh = MAX(High, 20)[1];
    double rangeLow = MIN(Low, 20)[1];
    double rangeMiddle = (rangeHigh + rangeLow) / 2;

    // Range bounce entry
    if (Close[1] < rangeLow && Close[0] > rangeLow && Position.MarketPosition ==
    {
        // Long entry at range bottom
        EnterLong();

        // Set volatility-based stop
        SetStopLoss(CalculationMode.Price, Low[0] - (atr[0] * 1.5));

        // Set profit target at range middle
        SetProfitTarget(CalculationMode.Price, rangeMiddle);
    }
    else if (Close[1] > rangeHigh && Close[0] < rangeHigh && Position.MarketPosi
    {
        // Short entry at range top
        EnterShort();

        // Set volatility-based stop
```

```
        SetStopLoss(CalculationMode.Price, High[0] + (atr[0] * 1.5));

        // Set profit target at range middle
        SetProfitTarget(CalculationMode.Price, rangeMiddle);
    }
}
```

Scalping with Tight Fixed Stops

Description: Quick entries and exits with tight fixed stops.

Example:

```
protected override void OnBarUpdate()
{
    // Wait for enough bars
    if (CurrentBar < 20)
        return;

    // Scalping entry
    if (CrossAbove(SMA(5), SMA(10), 1) && Position.MarketPosition == MarketPosit
    {
        EnterLong();

        // Set tight stop loss
        SetStopLoss(CalculationMode.Ticks, 5);

        // Set profit target
        SetProfitTarget(CalculationMode.Ticks, 10);
    }
}
```

Multi-Stage Exit Strategy

Description: Combines multiple exit techniques for comprehensive trade management.

Example:

```
private ATR atr;
private bool stageOneExitPlaced = false;
private bool stageTwoExitPlaced = false;

protected override void OnStateChange()
{
    if (State == State.Configure)
    {
        atr = ATR(14);
```

```
    }
}

protected override void OnBarUpdate()
{
    // Wait for enough bars
    if (CurrentBar < 20)
        return;

    // Entry logic
    if (CrossAbove(SMA(14), SMA(28), 1) && Position.MarketPosition == MarketPosi
    {
        EnterLong();

        // Initial stop loss
        SetStopLoss(CalculationMode.Price, Low[0] - (atr[0] * 1.5));

        stageOneExitPlaced = false;
        stageTwoExitPlaced = false;
    }

    // Multi-stage exit logic
    if (Position.MarketPosition == MarketPosition.Long)
    {
        // Stage One: Move to breakeven after 10 ticks profit
        if (!stageOneExitPlaced && Close[0] >= Position.AveragePrice + 10 * Tick
        {
            SetStopLoss(CalculationMode.Price, Position.AveragePrice + 2 * TickS
            stageOneExitPlaced = true;
        }

        // Stage Two: Trail stop after 20 ticks profit
        if (stageOneExitPlaced && !stageTwoExitPlaced && Close[0] >= Position.Av
        {
            SetTrailStop(CalculationMode.Ticks, 10);
            stageTwoExitPlaced = true;
        }

        // Stage Three: Scale out at 30 ticks profit
        if (stageTwoExitPlaced && Close[0] >= Position.AveragePrice + 30 * TickS
        {
            ExitLong(Position.Quantity / 2, "Partial Exit", "");
        }
    }
    else
    {
        stageOneExitPlaced = false;
        stageTwoExitPlaced = false;
    }
}
```

Troubleshooting Stop Orders

Common stop order issues and their solutions.

Stop Order Not Triggered

Common Causes: - Price gapped through stop level - Stop level too close to current price - Liquidity issues - Connection problems

Solutions: - Use market orders for stops in volatile markets - Set stops at a reasonable distance from current price - Consider using simulated stops for illiquid markets - Check connection status regularly

Stop Order Filled at Unexpected Price

Common Causes: - Slippage due to market volatility - Low liquidity - Fast-moving markets

Solutions: - Use limit orders for stops when possible - Adjust risk management to account for slippage - Trade more liquid markets - Consider using guaranteed stops if available

Stop Order Rejected

Common Causes: - Invalid parameters - Stop level too close to current price - Broker restrictions - Connection issues

Solutions: - Check order parameters - Ensure stop level is at a valid distance from current price - Review broker requirements for stop orders - Check connection status

Stop Order Not Updated

Common Causes: - Logic errors in code - Condition for update not met - Order ID tracking issues

Solutions: - Debug code with Print statements - Verify update conditions - Implement proper order ID tracking - Use OnOrderUpdate event for debugging

Multiple Stop Orders Created

Common Causes: - Missing order ID tracking - Duplicate order submission - Failure to cancel previous orders

Solutions: - Implement proper order ID tracking - Check for existing orders before submitting new ones - Cancel previous orders before submitting new ones - Use OCO (One-Cancels-Other) groups for related orders

This comprehensive guide covers all aspects of stop types and ATR trailing stops in NinjaTrader. For information on other aspects of NinjaTrader programming, please refer to the relevant sections in this knowledge base.

NinjaTrader API Reference and Advanced Topics

This comprehensive guide documents the NinjaTrader API and advanced programming topics for NinjaScript development.

Table of Contents

- Introduction to the NinjaTrader API
- API Architecture Overview
- Core API Components
- Common Methods and Properties
- Event-Driven Programming
- Advanced Data Handling
- Performance Optimization
- Debugging Techniques
- Multi-Timeframe Analysis
- Integration with External Data Sources
- Advanced Drawing Techniques
- Memory Management
- Common Design Patterns
- Troubleshooting and Best Practices

Introduction to the NinjaTrader API

The NinjaTrader API provides a comprehensive framework for developing custom trading applications, indicators, and strategies within the NinjaTrader platform. Built on C# and the .NET Framework, it offers a robust set of classes and methods specifically tailored for financial market analysis and trading system development.

API Structure

The NinjaTrader API is organized into several key namespaces:

- **NinjaTrader.Cbi**: Core business interfaces for market data and order management
- **NinjaTrader.Data**: Data management and manipulation
- **NinjaTrader.Gui**: User interface components
- **NinjaTrader.NinjaScript**: Core NinjaScript functionality
- **NinjaTrader.Core**: Core platform functionality
- **NinjaTrader.Custom**: Custom drawing and visualization

Development Environment

NinjaScript development is done within the NinjaTrader platform using the built-in NinjaScript Editor, which provides:

- Syntax highlighting

- IntelliSense code completion
- Error highlighting
- Integrated debugging
- Code snippets
- Project management

API Architecture Overview

The NinjaTrader API follows an object-oriented architecture with a strong emphasis on event-driven programming. Understanding this architecture is crucial for effective NinjaScript development.

Object Hierarchy

```
NinjaScriptBase
├── Indicator
│   ├── Custom Indicators
│   └── Built-in Indicators
├── Strategy
│   ├── Custom Strategies
│   └── Built-in Strategies
├── DrawingTool
│   ├── Custom Drawing Tools
│   └── Built-in Drawing Tools
├── BarsType
│   ├── Custom Bars Types
│   └── Built-in Bars Types
└── Other NinjaScript Types
```

State Management

NinjaScript objects follow a state machine pattern, transitioning through various states during their lifecycle:

1. **SetDefaults**: Initial state for setting default property values
2. **Configure**: Configuration state for adding indicators and setting up dependencies
3. **Active**: Active state where the object is processing data
4. **DataLoaded**: State indicating that data has been loaded
5. **Historical**: Processing historical data
6. **Transition**: Transitioning from historical to real-time
7. **Realtime**: Processing real-time data
8. **Terminated**: Object is being terminated

Understanding these states is essential for proper resource management and event handling.

Core API Components

NinjaScriptBase

The `NinjaScriptBase` class is the foundation of all NinjaScript objects, providing common functionality and properties.

```
public abstract class NinjaScriptBase
{
    // Core properties
    public string Name { get; set; }
    public string Description { get; set; }

    // State management
    public State State { get; }

    // Core methods
    protected virtual void OnStateChange();
    protected virtual void SetDefaults();
    protected virtual void Configure();
}
```

Indicator Class

The `Indicator` class extends `NinjaScriptBase` and provides functionality for creating custom indicators.

```
public class Indicator : NinjaScriptBase
{
    // Indicator-specific properties
    public bool IsOverlay { get; set; }
    public Calculate Calculate { get; set; }

    // Indicator-specific methods
    protected virtual void OnBarUpdate();
    protected virtual void OnMarketData(MarketDataEventArgs marketDataUpdate);

    // Plotting methods
    public void Plot(string plotName, PlotStyle plotStyle, Brush brush, Brush ar
    public void PlotColors(string plotName, Brush brush, Brush opacity);
}
```

Strategy Class

The `Strategy` class extends `NinjaScriptBase` and provides functionality for creating custom trading strategies.

```
public class Strategy : NinjaScriptBase
{
```

```
    // Strategy-specific properties
    public StrategyBase.OrderFillResolution OrderFillResolution { get; set; }
    public bool IsExitOnSessionCloseStrategy { get; set; }

    // Strategy-specific methods
    protected virtual void OnBarUpdate();
    protected virtual void OnExecutionUpdate(Execution execution, string executi
    protected virtual void OnOrderUpdate(Order order, double limitPrice, double

    // Order methods
    public void EnterLong();
    public void EnterShort();
    public void ExitLong();
    public void ExitShort();
    public void SetStopLoss(CalculationMode mode, double value);
    public void SetProfitTarget(CalculationMode mode, double value);
}
```

Common Methods and Properties

The NinjaTrader API provides a rich set of common methods and properties available to all NinjaScript types. These are essential building blocks for any NinjaScript development.

Data Access Methods

```
// Price data access
double Close[int barsAgo];
double Open[int barsAgo];
double High[int barsAgo];
double Low[int barsAgo];
double Volume[int barsAgo];
DateTime Time[int barsAgo];

// Indicator values
double SMA(int period)[int barsAgo];
double EMA(int period)[int barsAgo];
double RSI(int period)[int barsAgo];
```

Bar Management

```
// Bar information
int CurrentBar { get; }
int BarsInProgress { get; }
bool IsFirstTickOfBar { get; }
bool IsLastBarOfSession { get; }
```

```
// Bar manipulation
void AddDataSeries(string instrumentName, PeriodType periodType, int period);
void AddDataSeries(string instrumentName, PeriodType periodType, int period, Mar
```

Instrument Properties

```
// Instrument information
string Instrument.FullName { get; }
string Instrument.MasterInstrument.Name { get; }
double Instrument.MasterInstrument.PointValue { get; }
double Instrument.MasterInstrument.TickSize { get; }
```

Drawing Methods

```
// Drawing on charts
Draw.Line(string tag, int startBarsAgo, double startY, int endBarsAgo, double en
Draw.Text(string tag, string text, int barsAgo, double y, Brush brush);
Draw.ArrowUp(string tag, bool isAutoScale, int barsAgo, double y, Brush brush);
Draw.ArrowDown(string tag, bool isAutoScale, int barsAgo, double y, Brush brush)
```

Alert Methods

```
// Generating alerts
Alert(string id, Priority priority, string message, string soundLocation, int re
```

Event-Driven Programming

NinjaScript follows an event-driven programming model, where code execution is triggered by specific events. Understanding these events and their sequence is crucial for effective NinjaScript development.

Core Events

OnStateChange

The `OnStateChange` event is called whenever the state of a NinjaScript object changes. It's the primary method for initializing and configuring NinjaScript objects.

```
protected override void OnStateChange()
{
    if (State == State.SetDefaults)
    {
```

```
        Name = "MyCustomIndicator";
        Description = "A custom indicator example";
        Calculate = Calculate.OnBarClose;
        IsOverlay = false;
    }
    else if (State == State.Configure)
    {
        // Add indicators or other dependencies
        AddPlot(Brushes.DodgerBlue, "MyPlot");
    }
    else if (State == State.DataLoaded)
    {
        // Initialize variables after data is loaded
    }
    else if (State == State.Terminated)
    {
        // Clean up resources
    }
}
```

OnBarUpdate

The `OnBarUpdate` event is called for each bar update and is the primary method for implementing indicator and strategy logic.

```
protected override void OnBarUpdate()
{
    // Skip calculation until we have enough bars
    if (CurrentBar < 20)
        return;

    // Calculate indicator values
    double smaValue = SMA(20)[0];

    // Plot values
    Value[0] = smaValue;

    // For strategies, place orders
    if (Close[0] > smaValue && Close[1] <= smaValue[1])
    {
        EnterLong();
    }
}
```

Market Data Events

OnMarketData

The `OnMarketData` event is called when new market data is received, providing real-time price updates.

```
protected override void OnMarketData(MarketDataEventArgs marketDataUpdate)
{
    if (marketDataUpdate.MarketDataType == MarketDataType.Last)
    {
        // Process last price
        double lastPrice = marketDataUpdate.Price;

        // Take action based on last price
        if (lastPrice > someThreshold)
        {
            // Do something
        }
    }
}
```

OnMarketDepth

The `OnMarketDepth` event is called when market depth (Level II) data changes.

```
protected override void OnMarketDepth(MarketDepthEventArgs marketDepthUpdate)
{
    // Process market depth update
    if (marketDepthUpdate.MarketDataType == MarketDataType.Ask)
    {
        // Process ask side update
    }
    else if (marketDepthUpdate.MarketDataType == MarketDataType.Bid)
    {
        // Process bid side update
    }
}
```

Order and Execution Events

OnOrderUpdate

The `OnOrderUpdate` event is called when an order status changes.

```
protected override void OnOrderUpdate(Order order, double limitPrice, double stc
{
    if (orderState == OrderState.Filled)
    {
        // Order has been filled
        Print("Order filled: " + order.Name + " at price " + averageFillPrice);
```

```
    }
    else if (orderState == OrderState.Rejected)
    {
        // Order has been rejected
        Print("Order rejected: " + order.Name + " - " + error.ToString());
    }
}
```

OnExecutionUpdate

The `OnExecutionUpdate` event is called when an execution (fill) occurs.

```
protected override void OnExecutionUpdate(Execution execution, string executionI
{
    // Process execution
    Print("Execution: " + executionId + " - " + quantity + " @ " + price);

    // Take additional actions based on execution
    if (marketPosition == MarketPosition.Long)
    {
        // Long position established
    }
    else if (marketPosition == MarketPosition.Short)
    {
        // Short position established
    }
}
```

Other Events

OnFundamentalData

The `OnFundamentalData` event is called when fundamental data for an instrument changes.

```
protected override void OnFundamentalData(FundamentalDataEventArgs fundamentalDa
{
    // Process fundamental data
    if (fundamentalDataUpdate.FundamentalDataType == FundamentalDataType.Current
    {
        // Process current value
    }
}
```

OnConnectionStatusUpdate

The `OnConnectionStatusUpdate` event is called when the connection status to a data provider changes.

```
protected override void OnConnectionStatusUpdate(ConnectionStatusEventArgs conne
{
    if (connectionStatusUpdate.Status == ConnectionStatus.Connected)
    {
        // Connection established
        Print("Connected to " + connectionStatusUpdate.Provider.Name);
    }
    else if (connectionStatusUpdate.Status == ConnectionStatus.Disconnected)
    {
        // Connection lost
        Print("Disconnected from " + connectionStatusUpdate.Provider.Name);
    }
}
```

Advanced Data Handling

Effective data management is crucial for developing robust NinjaScript applications. This section covers advanced techniques for working with market data.

Multi-Series Data Management

NinjaScript supports working with multiple data series simultaneously, allowing for multi-timeframe analysis and cross-instrument strategies.

```
protected override void OnStateChange()
{
    if (State == State.Configure)
    {
        // Add a 5-minute data series of the primary instrument
        AddDataSeries(PeriodType.Minute, 5);

        // Add a daily data series of the primary instrument
        AddDataSeries(PeriodType.Day, 1);

        // Add a 1-minute data series of a different instrument
        AddDataSeries("ES 06-20", PeriodType.Minute, 1);
    }
}

protected override void OnBarUpdate()
{
    // Determine which BarsInProgress is calling OnBarUpdate()
    if (BarsInProgress == 0)
    {
```

```
        // Primary data series (could be any timeframe)
        Print("Primary series: " + Instrument.FullName + " " + BarsPeriod.Value
    }
    else if (BarsInProgress == 1)
    {
        // 5-minute data series
        Print("Secondary series: " + Instrument.FullName + " 5 Minute");
    }
    else if (BarsInProgress == 2)
    {
        // Daily data series
        Print("Third series: " + Instrument.FullName + " Daily");
    }
    else if (BarsInProgress == 3)
    {
        // Different instrument
        Print("Fourth series: ES 06-20 1 Minute");
    }
}
```

Custom Data Series

You can create custom data series to store calculated values for later reference.

```
private Series<double> customValues;

protected override void OnStateChange()
{
    if (State == State.SetDefaults)
    {
        Name = "CustomSeriesExample";
        Description = "Example of using custom data series";
    }
    else if (State == State.DataLoaded)
    {
        // Initialize custom series
        customValues = new Series<double>(this);
    }
}

protected override void OnBarUpdate()
{
    // Calculate and store a value
    customValues[0] = (High[0] + Low[0]) / 2;

    // Access previously calculated values
    if (CurrentBar > 0)
    {
        double previousValue = customValues[1];
```

```
        Print("Previous value: " + previousValue);
    }
}
```

Market Analyzer Data

NinjaScript allows you to create custom columns for the Market Analyzer, providing real-time data analysis across multiple instruments.

```
public class CustomMAColumn : MarketAnalyzerColumn
{
    private SMA sma;

    protected override void OnStateChange()
    {
        if (State == State.SetDefaults)
        {
            Name = "Custom SMA";
            Description = "Displays the SMA value for each instrument";
        }
        else if (State == State.Configure)
        {
            sma = SMA(14);
        }
    }

    protected override void OnMarketData(MarketDataEventArgs marketDataUpdate)
    {
        if (marketDataUpdate.MarketDataType == MarketDataType.Last)
        {
            // Update the column value with the current SMA
            CurrentValue = sma[0];
        }
    }
}
```

Historical Data Processing

Working with historical data requires special consideration, especially for backtesting strategies.

```
protected override void OnBarUpdate()
{
    // Check if we're processing historical data
    if (State == State.Historical)
    {
        // Processing historical data (backtesting)
        // Avoid resource-intensive operations
    }
```

```
    else if (State == State.Realtime)
    {
        // Processing real-time data
        // Can perform more intensive operations
    }

    // Common calculations for both historical and real-time
    double smaValue = SMA(20)[0];

    // Take action based on calculations
    if (CrossAbove(Close, smaValue, 1))
    {
        // Bullish signal
        if (State == State.Realtime)
        {
            // Generate alert only in real-time
            Alert("SMA_CrossAbove", Priority.High, "Price crossed above SMA", ""
        }

        // Enter position (works in both historical and real-time)
        EnterLong();
    }
}
```

Performance Optimization

Optimizing NinjaScript code is essential for efficient execution, especially for strategies running in real-time or processing large amounts of historical data.

Calculation Optimization

```
// Inefficient approach - recalculates SMA for each bar
protected override void OnBarUpdate()
{
    for (int i = 0; i < 10; i++)
    {
        double smaValue = SMA(20)[i]; // Recalculates SMA multiple times
        // Use smaValue
    }
}

// Efficient approach - calculates SMA once and stores values
protected override void OnBarUpdate()
{
    double[] smaValues = new double[10];
    for (int i = 0; i < 10; i++)
    {
        smaValues[i] = SMA(20)[i]; // Still not optimal
```

```
    }

    // Even better approach
    SMA sma = SMA(20); // Reference the indicator once
    for (int i = 0; i < 10; i++)
    {
        double smaValue = sma[i]; // Access values from the reference
        // Use smaValue
    }
}
```

Memory Management

```
// Poor memory management
private List<double> priceHistory = new List<double>(); // Unbounded growth

protected override void OnBarUpdate()
{
    // Adding to the list without bounds will cause memory issues over time
    priceHistory.Add(Close[0]);

    // Process the list
    // ...
}

// Better memory management
private Queue<double> priceHistory = new Queue<double>(100); // Fixed size

protected override void OnBarUpdate()
{
    // Maintain a fixed size collection
    if (priceHistory.Count >= 100)
        priceHistory.Dequeue(); // Remove oldest item

    priceHistory.Enqueue(Close[0]); // Add newest item

    // Process the queue
    // ...
}
```

Drawing Optimization

```
// Inefficient drawing - recreates all objects on each bar
protected override void OnBarUpdate()
{
    // Clear all drawings
    RemoveDrawObjects();
```

```
    // Redraw everything
    for (int i = 0; i < 20; i++)
    {
        Draw.Line("line" + i, i, Low[i], i, High[i], Brushes.Blue);
    }
}

// Efficient drawing - updates only what's needed
protected override void OnBarUpdate()
{
    // Only draw the current bar
    Draw.Line("line" + CurrentBar, 0, Low[0], 0, High[0], Brushes.Blue);

    // Remove old drawings if needed
    if (CurrentBar > 100)
    {
        RemoveDrawObject("line" + (CurrentBar - 100));
    }
}
```

Conditional Execution

```
// Inefficient - performs calculations on every bar
protected override void OnBarUpdate()
{
    // Complex calculations
    double complexValue = PerformComplexCalculation();

    // Only needed in specific conditions
    if (SomeRareCondition())
    {
        UseComplexValue(complexValue);
    }
}

// Efficient - performs calculations only when needed
protected override void OnBarUpdate()
{
    // Only perform complex calculations when needed
    if (SomeRareCondition())
    {
        double complexValue = PerformComplexCalculation();
        UseComplexValue(complexValue);
    }
}
```

Debugging Techniques

Effective debugging is essential for developing reliable NinjaScript applications. NinjaTrader provides several tools and techniques for debugging.

Print Statements

The simplest debugging technique is to use `Print` statements to output values to the NinjaTrader Control Center's Output tab.

```
protected override void OnBarUpdate()
{
    // Output basic information
    Print("Bar #" + CurrentBar + " - Open: " + Open[0] + ", High: " + High[0] +

    // Debug calculated values
    double smaValue = SMA(20)[0];
    Print("SMA(20): " + smaValue);

    // Conditional debugging
    if (CrossAbove(Close, smaValue, 1))
    {
        Print("CrossAbove detected at bar " + CurrentBar);
    }
}
```

Trace Levels

NinjaTrader supports different trace levels for more controlled debugging output.

```
protected override void OnBarUpdate()
{
    // Basic information (always shown)
    Trace(0, "Bar #" + CurrentBar + " processing");

    // Detailed information (shown only when trace level >= 1)
    Trace(1, "SMA(20): " + SMA(20)[0]);

    // Very detailed information (shown only when trace level >= 2)
    Trace(2, "Full bar data - Open: " + Open[0] + ", High: " + High[0] + ", Low:
}
```

Visual Debugging

Visual debugging techniques can be more intuitive than text-based debugging.

```
protected override void OnBarUpdate()
{
```

```
    // Draw values on the chart for debugging
    Draw.Text("Debug" + CurrentBar, "SMA: " + SMA(20)[0].ToString("0.00"), 0, Lo

    // Highlight important bars
    if (SomeCondition())
    {
        Draw.Diamond("Important" + CurrentBar, 0, High[0] + 5 * TickSize, Brushe
    }

    // Visualize calculations
    double upperBand = SMA(20)[0] + 2 * StdDev(20)[0];
    double lowerBand = SMA(20)[0] - 2 * StdDev(20)[0];

    Draw.Line("UpperBand" + CurrentBar, 1, upperBand, 0, upperBand, Brushes.Blue
    Draw.Line("LowerBand" + CurrentBar, 1, lowerBand, 0, lowerBand, Brushes.Blue
}
```

Debugging Strategies

Strategies require special debugging techniques, especially for order-related issues.

```
protected override void OnOrderUpdate(Order order, double limitPrice, double sto
{
    // Log all order updates
    Print("Order Update: " + order.Name + " - State: " + orderState + " - Error:

    // Detailed logging for specific states
    if (orderState == OrderState.Rejected)
    {
        Print("Order Rejected: " + order.Name + " - Error: " + error + " - Comme

        // Log additional context
        Print("Market position: " + Position.MarketPosition);
        Print("Current price: " + Close[0]);
        Print("Account value: " + Account.Get(AccountItem.CashValue, Currency.Us
    }
}

protected override void OnExecutionUpdate(Execution execution, string executionI
{
    // Log all executions
    Print("Execution: " + executionId + " - Price: " + price + " - Quantity: " +
}
```

Multi-Timeframe Analysis

Multi-timeframe analysis is a powerful technique for developing more robust trading systems.

NinjaTrader provides several methods for implementing multi-timeframe strategies.

Using Multiple Data Series

```
private SMA smaDaily;
private SMA smaHourly;
private SMA smaPrimary;

protected override void OnStateChange()
{
    if (State == State.SetDefaults)
    {
        Name = "MultiTimeframeStrategy";
        Description = "Strategy using multiple timeframes";
    }
    else if (State == State.Configure)
    {
        // Add higher timeframe data series
        AddDataSeries(PeriodType.Minute, 60); // Hourly
        AddDataSeries(PeriodType.Day, 1);     // Daily

        // Create indicators for each timeframe
        smaPrimary = SMA(20);
        smaHourly = SMA(20);
        smaDaily = SMA(20);
    }
    else if (State == State.DataLoaded)
    {
        // Assign indicators to specific BarsInProgress
        smaHourly.BarsInProgress = 1; // Hourly data
        smaDaily.BarsInProgress = 2;  // Daily data
    }
}

protected override void OnBarUpdate()
{
    // Skip if not enough bars
    if (CurrentBar < 20)
        return;

    if (BarsInProgress == 0) // Primary timeframe
    {
        // Access indicator values from different timeframes
        double primarySMA = smaPrimary[0];
        double hourlySMA = smaHourly[0];
        double dailySMA = smaDaily[0];

        // Trading logic using multiple timeframes
        if (Close[0] > dailySMA && Close[0] > hourlySMA && CrossAbove(Close, pri
        {
```

```
            EnterLong();
        }
        else if (Close[0] < dailySMA && Close[0] < hourlySMA && CrossBelow(Close
        {
            EnterShort();
        }
    }
}
```

TimeFrame Indicator

NinjaTrader provides a `TimeFrame` indicator for accessing data from different timeframes.

```
private TimeFrame tfDaily;
private TimeFrame tfHourly;

protected override void OnStateChange()
{
    if (State == State.SetDefaults)
    {
        Name = "TimeFrameExample";
        Description = "Example using TimeFrame indicator";
    }
    else if (State == State.Configure)
    {
        // Create TimeFrame indicators
        tfHourly = TimeFrame(PeriodType.Minute, 60);
        tfDaily = TimeFrame(PeriodType.Day, 1);
    }
}

protected override void OnBarUpdate()
{
    // Skip if not enough bars
    if (CurrentBar < 20)
        return;

    // Access data from different timeframes
    double hourlyOpen = tfHourly.Open[0];
    double hourlyHigh = tfHourly.High[0];
    double hourlyLow = tfHourly.Low[0];
    double hourlyClose = tfHourly.Close[0];

    double dailyOpen = tfDaily.Open[0];
    double dailyHigh = tfDaily.High[0];
    double dailyLow = tfDaily.Low[0];
    double dailyClose = tfDaily.Close[0];

    // Access indicators on different timeframes
```

```
    double hourlySMA = tfHourly.SMA(20)[0];
    double dailySMA = tfDaily.SMA(20)[0];

    // Trading logic using multiple timeframes
    if (Close[0] > dailySMA && CrossAbove(Close, hourlySMA, 1))
    {
        EnterLong();
    }
}
```

Synchronizing Data

When working with multiple timeframes, it's important to ensure that data is properly synchronized.

```
protected override void OnBarUpdate()
{
    // Skip if not enough bars
    if (CurrentBar < 20)
        return;

    if (BarsInProgress == 0) // Primary timeframe
    {
        // Check if we have data for all timeframes
        if (BarsArray[1].Count <= 0 || BarsArray[2].Count <= 0)
            return;

        // Check if higher timeframe data is current
        DateTime primaryTime = Time[0];
        DateTime hourlyTime = Times[1][0];
        DateTime dailyTime = Times[2][0];

        // Ensure the higher timeframe bars contain the current bar
        bool hourlyContainsPrimary = primaryTime.Date == hourlyTime.Date &&
                                     primaryTime.Hour == hourlyTime.Hour;

        bool dailyContainsPrimary = primaryTime.Date == dailyTime.Date;

        if (!hourlyContainsPrimary || !dailyContainsPrimary)
            return;

        // Now we can safely use data from all timeframes
        // ...
    }
}
```

Integration with External Data Sources

NinjaTrader can be integrated with external data sources to enhance trading strategies with additional information.

Using External Files

```
private Dictionary<DateTime, double> externalData;

protected override void OnStateChange()
{
    if (State == State.SetDefaults)
    {
        Name = "ExternalDataExample";
        Description = "Example of using external data";
    }
    else if (State == State.DataLoaded)
    {
        // Load external data
        externalData = LoadExternalData("C:\\Data\\economic_data.csv");
    }
}

private Dictionary<DateTime, double> LoadExternalData(string filePath)
{
    Dictionary<DateTime, double> data = new Dictionary<DateTime, double>();

    try
    {
        string[] lines = File.ReadAllLines(filePath);

        foreach (string line in lines)
        {
            string[] parts = line.Split(',');
            if (parts.Length >= 2)
            {
                DateTime date;
                double value;

                if (DateTime.TryParse(parts[0], out date) && double.TryParse(par
                {
                    data[date] = value;
                }
            }
        }
    }
    catch (Exception ex)
    {
        Print("Error loading external data: " + ex.Message);
    }

    return data;
```

```
}

protected override void OnBarUpdate()
{
    // Skip if not enough bars
    if (CurrentBar < 20)
        return;

    // Get the date for the current bar
    DateTime currentDate = Time[0].Date;

    // Check if we have external data for this date
    if (externalData.ContainsKey(currentDate))
    {
        double externalValue = externalData[currentDate];

        // Use the external data in your strategy
        if (externalValue > someThreshold && Close[0] > SMA(20)[0])
        {
            EnterLong();
        }
    }
}
```

Using Web Services

```
private WebClient webClient;
private string apiKey = "your_api_key";
private string apiUrl = "https://api.example.com/data";
private Dictionary<DateTime, double> apiData;

protected override void OnStateChange()
{
    if (State == State.SetDefaults)
    {
        Name = "WebServiceExample";
        Description = "Example of using web services";
    }
    else if (State == State.DataLoaded)
    {
        // Initialize web client
        webClient = new WebClient();
        apiData = new Dictionary<DateTime, double>();

        // Set up event handler for completed downloads
        webClient.DownloadStringCompleted += OnDownloadCompleted;

        // Start initial data download
```

```
        FetchData();
    }
    else if (State == State.Terminated)
    {
        // Clean up resources
        if (webClient != null)
        {
            webClient.Dispose();
            webClient = null;
        }
    }
}

private void FetchData()
{
    try
    {
        // Build the API request URL
        string requestUrl = apiUrl + "?key=" + apiKey + "&date=" + DateTime.Now.

        // Make the asynchronous request
        webClient.DownloadStringAsync(new Uri(requestUrl));
    }
    catch (Exception ex)
    {
        Print("Error fetching data: " + ex.Message);
    }
}

private void OnDownloadCompleted(object sender, DownloadStringCompletedEventArgs
{
    if (e.Error != null)
    {
        Print("Download error: " + e.Error.Message);
        return;
    }

    if (e.Cancelled)
    {
        Print("Download cancelled");
        return;
    }

    try
    {
        // Parse the JSON response
        JObject json = JObject.Parse(e.Result);

        // Extract data
        foreach (JProperty property in json["data"].Children<JProperty>())
        {
```

```
            DateTime date;
            if (DateTime.TryParse(property.Name, out date))
            {
                double value = property.Value.Value<double>();
                apiData[date] = value;
            }
        }

        Print("Data updated successfully");
    }
    catch (Exception ex)
    {
        Print("Error parsing data: " + ex.Message);
    }
}

protected override void OnBarUpdate()
{
    // Skip if not enough bars
    if (CurrentBar < 20)
        return;

    // Get the date for the current bar
    DateTime currentDate = Time[0].Date;

    // Check if we have API data for this date
    if (apiData.ContainsKey(currentDate))
    {
        double apiValue = apiData[currentDate];

        // Use the API data in your strategy
        if (apiValue > someThreshold && Close[0] > SMA(20)[0])
        {
            EnterLong();
        }
    }

    // Periodically refresh data (e.g., once per day)
    if (IsFirstTickOfBar && Time[0].Hour == 0 && Time[0].Minute == 0)
    {
        FetchData();
    }
}
```

Using DLL Imports

```
// Import external functions from a DLL
[DllImport("ExternalLibrary.dll")]
```

```
private static extern double CalculateIndicator(double[] prices, int length);

[DllImport("ExternalLibrary.dll")]
private static extern int GetSignal(double[] prices, double[] volumes, int lengt

private double[] priceBuffer;
private double[] volumeBuffer;
private int bufferSize = 50;

protected override void OnStateChange()
{
    if (State == State.SetDefaults)
    {
        Name = "DllImportExample";
        Description = "Example of using DLL imports";
    }
    else if (State == State.DataLoaded)
    {
        // Initialize buffers
        priceBuffer = new double[bufferSize];
        volumeBuffer = new double[bufferSize];
    }
}

protected override void OnBarUpdate()
{
    // Skip if not enough bars
    if (CurrentBar < bufferSize)
        return;

    // Fill the buffers with recent data
    for (int i = 0; i < bufferSize; i++)
    {
        priceBuffer[i] = Close[i];
        volumeBuffer[i] = Volume[i];
    }

    // Call external functions
    double indicatorValue = CalculateIndicator(priceBuffer, bufferSize);
    int signal = GetSignal(priceBuffer, volumeBuffer, bufferSize);

    // Use the results in your strategy
    if (signal > 0)
    {
        EnterLong();
    }
    else if (signal < 0)
    {
        EnterShort();
    }
```

```
    // Plot the indicator value
    Value[0] = indicatorValue;
}
```

Advanced Drawing Techniques

NinjaTrader provides powerful drawing capabilities for creating custom visualizations on charts.

Custom Drawing Tools

```
public class CustomDrawingTool : DrawingTool
{
    private Brush brush;
    private int lineWidth;

    protected override void OnStateChange()
    {
        if (State == State.SetDefaults)
        {
            Name = "Custom Drawing Tool";
            Description = "A custom drawing tool example";
        }
    }

    public override void OnRender(ChartControl chartControl, ChartScale chartSca
    {
        // Get the device context for drawing
        SharpDX.Direct2D1.RenderTarget renderTarget = chartControl.ChartPanel.D2

        // Create a brush for drawing
        SharpDX.Direct2D1.SolidColorBrush drawBrush = new SharpDX.Direct2D1.Soli

        // Get the points to draw
        Point startPoint = chartControl.ConvertToPixels(StartAnchor.Time, StartA
        Point endPoint = chartControl.ConvertToPixels(EndAnchor.Time, EndAnchor.

        // Draw a line
        renderTarget.DrawLine(
            new SharpDX.Vector2(startPoint.X, startPoint.Y),
            new SharpDX.Vector2(endPoint.X, endPoint.Y),
            drawBrush,
            2f
        );

        // Clean up resources
        drawBrush.Dispose();
    }
}
```

Custom Chart Rendering

```
protected override void OnRender(ChartControl chartControl, ChartScale chartScal
{
    // Call the base implementation
    base.OnRender(chartControl, chartScale);

    // Get the device context for drawing
    SharpDX.Direct2D1.RenderTarget renderTarget = chartControl.ChartPanel.D2DRen

    // Create brushes for drawing
    SharpDX.Direct2D1.SolidColorBrush lineBrush = new SharpDX.Direct2D1.SolidCol
    SharpDX.Direct2D1.SolidColorBrush fillBrush = new SharpDX.Direct2D1.SolidCol

    try
    {
        // Create a path geometry for the area
        SharpDX.Direct2D1.PathGeometry pathGeometry = new SharpDX.Direct2D1.Path
        SharpDX.Direct2D1.GeometrySink sink = pathGeometry.Open();

        // Start the path
        bool pathStarted = false;

        // Loop through visible bars
        for (int i = ChartBars.FromIndex; i <= ChartBars.ToIndex; i++)
        {
            if (i < 0 || i >= Plots[0].Values.Count)
                continue;

            // Get the upper and lower values
            double upperValue = Plots[0].Values[i];
            double lowerValue = Plots[1].Values[i];

            // Convert to screen coordinates
            Point upperPoint = chartControl.ConvertToPixels(i, upperValue);
            Point lowerPoint = chartControl.ConvertToPixels(i, lowerValue);

            // Start the path if not started
            if (!pathStarted)
            {
                sink.BeginFigure(new SharpDX.Vector2(upperPoint.X, upperPoint.Y)
                pathStarted = true;
            }

            // Add the upper point to the path
            sink.AddLine(new SharpDX.Vector2(upperPoint.X, upperPoint.Y));
        }
```

```
        // Add the lower points in reverse order
        for (int i = ChartBars.ToIndex; i >= ChartBars.FromIndex; i--)
        {
            if (i < 0 || i >= Plots[0].Values.Count)
                continue;

            // Get the lower value
            double lowerValue = Plots[1].Values[i];

            // Convert to screen coordinates
            Point lowerPoint = chartControl.ConvertToPixels(i, lowerValue);

            // Add the lower point to the path
            sink.AddLine(new SharpDX.Vector2(lowerPoint.X, lowerPoint.Y));
        }

        // Close the path
        if (pathStarted)
        {
            sink.EndFigure(SharpDX.Direct2D1.FigureEnd.Closed);
        }

        // Close the sink
        sink.Close();

        // Draw the filled area
        renderTarget.FillGeometry(pathGeometry, fillBrush);

        // Draw the upper and lower lines
        for (int i = ChartBars.FromIndex; i < ChartBars.ToIndex; i++)
        {
            if (i < 0 || i + 1 >= Plots[0].Values.Count)
                continue;

            // Get the upper values
            double upperValue1 = Plots[0].Values[i];
            double upperValue2 = Plots[0].Values[i + 1];

            // Get the lower values
            double lowerValue1 = Plots[1].Values[i];
            double lowerValue2 = Plots[1].Values[i + 1];

            // Convert to screen coordinates
            Point upperPoint1 = chartControl.ConvertToPixels(i, upperValue1);
            Point upperPoint2 = chartControl.ConvertToPixels(i + 1, upperValue2)

            Point lowerPoint1 = chartControl.ConvertToPixels(i, lowerValue1);
            Point lowerPoint2 = chartControl.ConvertToPixels(i + 1, lowerValue2)

            // Draw the upper line
            renderTarget.DrawLine(
```

```
                new SharpDX.Vector2(upperPoint1.X, upperPoint1.Y),
                new SharpDX.Vector2(upperPoint2.X, upperPoint2.Y),
                lineBrush,
                2f
            );

            // Draw the lower line
            renderTarget.DrawLine(
                new SharpDX.Vector2(lowerPoint1.X, lowerPoint1.Y),
                new SharpDX.Vector2(lowerPoint2.X, lowerPoint2.Y),
                lineBrush,
                2f
            );
        }
    }
    finally
    {
        // Clean up resources
        lineBrush.Dispose();
        fillBrush.Dispose();
    }
}
```

Interactive Drawing

```
private bool isDrawing;
private int startBar;
private double startPrice;
private int endBar;
private double endPrice;

protected override void OnRender(ChartControl chartControl, ChartScale chartScal
{
    // Call the base implementation
    base.OnRender(chartControl, chartScale);

    // Only draw if we're in drawing mode
    if (isDrawing)
    {
        // Get the device context for drawing
        SharpDX.Direct2D1.RenderTarget renderTarget = chartControl.ChartPanel.D2

        // Create a brush for drawing
        SharpDX.Direct2D1.SolidColorBrush drawBrush = new SharpDX.Direct2D1.Soli

        try
        {
            // Convert to screen coordinates
```

```
            Point startPoint = chartControl.ConvertToPixels(startBar, startPrice
            Point endPoint = chartControl.ConvertToPixels(endBar, endPrice);

            // Draw a line
            renderTarget.DrawLine(
                new SharpDX.Vector2(startPoint.X, startPoint.Y),
                new SharpDX.Vector2(endPoint.X, endPoint.Y),
                drawBrush,
                2f
            );
        }
        finally
        {
            // Clean up resources
            drawBrush.Dispose();
        }
    }
}

protected override void OnMouseDown(ChartControl chartControl, ChartPanel chartP
{
    // Start drawing
    isDrawing = true;
    startBar = dataPoint.DrawnOnBar;
    startPrice = dataPoint.Price;
    endBar = startBar;
    endPrice = startPrice;
}

protected override void OnMouseMove(ChartControl chartControl, ChartPanel chartP
{
    // Update end point if drawing
    if (isDrawing)
    {
        endBar = dataPoint.DrawnOnBar;
        endPrice = dataPoint.Price;

        // Force a redraw
        chartControl.Invalidate();
    }
}

protected override void OnMouseUp(ChartControl chartControl, ChartPanel chartPan
{
    // Finish drawing
    if (isDrawing)
    {
        endBar = dataPoint.DrawnOnBar;
        endPrice = dataPoint.Price;

        // Create a permanent drawing
```

```
        Draw.Line(
            "CustomLine" + CurrentBar,
            startBar,
            startPrice,
            endBar,
            endPrice,
            Brushes.Red
        );

        // Reset drawing state
        isDrawing = false;

        // Force a redraw
        chartControl.Invalidate();
    }
}
```

Memory Management

Proper memory management is crucial for developing efficient and stable NinjaScript applications.

Resource Cleanup

```
private WebClient webClient;
private Timer timer;
private List<SharpDX.Direct2D1.Brush> brushes;

protected override void OnStateChange()
{
    if (State == State.SetDefaults)
    {
        Name = "ResourceCleanupExample";
        Description = "Example of proper resource cleanup";
    }
    else if (State == State.Configure)
    {
        // Initialize resources
        webClient = new WebClient();
        timer = new Timer(60000); // 60-second timer
        timer.Elapsed += OnTimerElapsed;
        timer.Start();

        brushes = new List<SharpDX.Direct2D1.Brush>();
    }
    else if (State == State.Terminated)
    {
        // Clean up resources
```

```
        if (webClient != null)
        {
            webClient.Dispose();
            webClient = null;
        }

        if (timer != null)
        {
            timer.Stop();
            timer.Elapsed -= OnTimerElapsed;
            timer.Dispose();
            timer = null;
        }

        if (brushes != null)
        {
            foreach (SharpDX.Direct2D1.Brush brush in brushes)
            {
                if (brush != null)
                    brush.Dispose();
            }

            brushes.Clear();
            brushes = null;
        }
    }
}

private void OnTimerElapsed(object sender, ElapsedEventArgs e)
{
    // Timer event handler
}

protected override void OnRender(ChartControl chartControl, ChartScale chartScal
{
    // Call the base implementation
    base.OnRender(chartControl, chartScale);

    // Get the device context for drawing
    SharpDX.Direct2D1.RenderTarget renderTarget = chartControl.ChartPanel.D2DRen

    // Create a brush for drawing
    SharpDX.Direct2D1.SolidColorBrush drawBrush = new SharpDX.Direct2D1.SolidCol

    // Add to the list for cleanup
    brushes.Add(drawBrush);

    // Use the brush for drawing
    // ...
}
```

Memory Leaks Prevention

```
// Potential memory leak
private List<double> dataPoints = new List<double>();

protected override void OnBarUpdate()
{
    // Adding data without bounds
    dataPoints.Add(Close[0]);

    // Using the data
    // ...
}

// Memory leak prevention
private Queue<double> dataPoints = new Queue<double>(100); // Fixed size

protected override void OnBarUpdate()
{
    // Maintain a fixed size collection
    if (dataPoints.Count >= 100)
        dataPoints.Dequeue(); // Remove oldest item

    dataPoints.Enqueue(Close[0]); // Add newest item

    // Using the data
    // ...
}
```

Event Handler Management

```
private Timer timer;

protected override void OnStateChange()
{
    if (State == State.SetDefaults)
    {
        Name = "EventHandlerExample";
        Description = "Example of proper event handler management";
    }
    else if (State == State.Configure)
    {
        // Initialize timer
        timer = new Timer(60000); // 60-second timer
        timer.Elapsed += OnTimerElapsed; // Add event handler
        timer.Start();
    }
```

```
    else if (State == State.Terminated)
    {
        // Clean up timer
        if (timer != null)
        {
            timer.Stop();
            timer.Elapsed -= OnTimerElapsed; // Remove event handler
            timer.Dispose();
            timer = null;
        }
    }
}

private void OnTimerElapsed(object sender, ElapsedEventArgs e)
{
    // Timer event handler
}
```

Common Design Patterns

Implementing common design patterns can improve the structure and maintainability of NinjaScript applications.

Strategy Template Pattern

```
public class StrategyTemplate : Strategy
{
    protected override void OnStateChange()
    {
        if (State == State.SetDefaults)
        {
            Name = "Strategy Template";
            Description = "A template for creating strategies";
        }
    }

    protected override void OnBarUpdate()
    {
        // Skip if not enough bars
        if (CurrentBar < 20)
            return;

        // Entry logic
        if (ShouldEnterLong())
        {
            EnterLong();
        }
        else if (ShouldEnterShort())
```

```
        {
            EnterShort();
        }

        // Exit logic
        if (ShouldExitLong())
        {
            ExitLong();
        }
        else if (ShouldExitShort())
        {
            ExitShort();
        }
    }

    // Template methods to be overridden by derived classes
    protected virtual bool ShouldEnterLong()
    {
        // Default implementation
        return CrossAbove(SMA(10), SMA(20), 1);
    }

    protected virtual bool ShouldEnterShort()
    {
        // Default implementation
        return CrossBelow(SMA(10), SMA(20), 1);
    }

    protected virtual bool ShouldExitLong()
    {
        // Default implementation
        return CrossBelow(SMA(10), SMA(20), 1);
    }

    protected virtual bool ShouldExitShort()
    {
        // Default implementation
        return CrossAbove(SMA(10), SMA(20), 1);
    }
}

// Derived strategy implementing the template
public class MovingAverageCrossover : StrategyTemplate
{
    private int fastPeriod = 10;
    private int slowPeriod = 20;

    protected override void OnStateChange()
    {
        if (State == State.SetDefaults)
        {
```

```
            Name = "Moving Average Crossover";
            Description = "A strategy based on moving average crossovers";

            // Add parameters
            fastPeriod = 10;
            slowPeriod = 20;
        }

        // Call base implementation
        base.OnStateChange();
    }

    // Override template methods
    protected override bool ShouldEnterLong()
    {
        return CrossAbove(SMA(fastPeriod), SMA(slowPeriod), 1);
    }

    protected override bool ShouldEnterShort()
    {
        return CrossBelow(SMA(fastPeriod), SMA(slowPeriod), 1);
    }

    protected override bool ShouldExitLong()
    {
        return CrossBelow(SMA(fastPeriod), SMA(slowPeriod), 1);
    }

    protected override bool ShouldExitShort()
    {
        return CrossAbove(SMA(fastPeriod), SMA(slowPeriod), 1);
    }
}
```

Observer Pattern

```
// Observer interface
public interface IMarketObserver
{
    void Update(double price, double volume);
}

// Concrete observer
public class MarketAnalyzer : IMarketObserver
{
    private double lastPrice;
    private double lastVolume;

    public void Update(double price, double volume)
```

```
    {
        lastPrice = price;
        lastVolume = volume;

        // Analyze the market data
        Analyze();
    }

    private void Analyze()
    {
        // Perform analysis
        // ...
    }
}

// Subject
public class MarketDataSubject : Indicator
{
    private List<IMarketObserver> observers = new List<IMarketObserver>();

    protected override void OnStateChange()
    {
        if (State == State.SetDefaults)
        {
            Name = "Market Data Subject";
            Description = "Provides market data to observers";
        }
    }

    protected override void OnMarketData(MarketDataEventArgs marketDataUpdate)
    {
        if (marketDataUpdate.MarketDataType == MarketDataType.Last)
        {
            // Notify all observers
            NotifyObservers(marketDataUpdate.Price, marketDataUpdate.Volume);
        }
    }

    public void AddObserver(IMarketObserver observer)
    {
        observers.Add(observer);
    }

    public void RemoveObserver(IMarketObserver observer)
    {
        observers.Remove(observer);
    }

    private void NotifyObservers(double price, double volume)
    {
        foreach (IMarketObserver observer in observers)
```

```
        {
            observer.Update(price, volume);
        }
    }
}

// Usage
public class ObserverPatternExample : Indicator
{
    private MarketDataSubject marketDataSubject;
    private MarketAnalyzer marketAnalyzer;

    protected override void OnStateChange()
    {
        if (State == State.SetDefaults)
        {
            Name = "Observer Pattern Example";
            Description = "Example of the Observer pattern";
        }
        else if (State == State.Configure)
        {
            // Create subject and observer
            marketDataSubject = new MarketDataSubject();
            marketAnalyzer = new MarketAnalyzer();

            // Register observer
            marketDataSubject.AddObserver(marketAnalyzer);
        }
        else if (State == State.Terminated)
        {
            // Unregister observer
            marketDataSubject.RemoveObserver(marketAnalyzer);
        }
    }
}
```

Strategy Pattern

```
// Strategy interface
public interface ITradingStrategy
{
    bool ShouldEnter(double price, double[] indicators);
    bool ShouldExit(double price, double[] indicators);
}

// Concrete strategies
public class MovingAverageCrossStrategy : ITradingStrategy
{
    public bool ShouldEnter(double price, double[] indicators)
```

```
    {
        // indicators[0] = fast MA, indicators[1] = slow MA
        return price > indicators[0] && indicators[0] > indicators[1];
    }

    public bool ShouldExit(double price, double[] indicators)
    {
        // indicators[0] = fast MA, indicators[1] = slow MA
        return price < indicators[0] && indicators[0] < indicators[1];
    }
}

public class RSIStrategy : ITradingStrategy
{
    public bool ShouldEnter(double price, double[] indicators)
    {
        // indicators[0] = RSI
        return indicators[0] < 30;
    }

    public bool ShouldExit(double price, double[] indicators)
    {
        // indicators[0] = RSI
        return indicators[0] > 70;
    }
}

// Context
public class TradingSystem : Strategy
{
    private ITradingStrategy tradingStrategy;
    private SMA fastMA;
    private SMA slowMA;
    private RSI rsi;

    protected override void OnStateChange()
    {
        if (State == State.SetDefaults)
        {
            Name = "Trading System";
            Description = "A trading system using the Strategy pattern";
        }
        else if (State == State.Configure)
        {
            // Initialize indicators
            fastMA = SMA(10);
            slowMA = SMA(20);
            rsi = RSI(14);

            // Set default strategy
            SetStrategy(new MovingAverageCrossStrategy());
```

```
        }
    }

    protected override void OnBarUpdate()
    {
        // Skip if not enough bars
        if (CurrentBar < 20)
            return;

        // Prepare indicators
        double[] indicators;

        if (tradingStrategy is MovingAverageCrossStrategy)
        {
            indicators = new double[] { fastMA[0], slowMA[0] };
        }
        else if (tradingStrategy is RSIStrategy)
        {
            indicators = new double[] { rsi[0] };
        }
        else
        {
            indicators = new double[0];
        }

        // Check entry conditions
        if (tradingStrategy.ShouldEnter(Close[0], indicators))
        {
            EnterLong();
        }

        // Check exit conditions
        if (tradingStrategy.ShouldExit(Close[0], indicators))
        {
            ExitLong();
        }
    }

    public void SetStrategy(ITradingStrategy strategy)
    {
        tradingStrategy = strategy;
    }
}
```

Troubleshooting and Best Practices

This section provides guidance on troubleshooting common issues and following best practices in NinjaScript development.

Common Issues and Solutions

Issue: Indicator Not Plotting

Symptoms: - Indicator compiles successfully but doesn't display on the chart - No error messages are shown

Possible Causes: - Plot method not called - Incorrect plot name - Values not assigned to the plot - IsOverlay property set incorrectly

Solutions:

```
protected override void OnStateChange()
{
    if (State == State.SetDefaults)
    {
        Name = "MyIndicator";
        Description = "My custom indicator";

        // Set IsOverlay based on whether the indicator should overlay the price
        IsOverlay = true; // or false

        // Add plots
        AddPlot(Brushes.DodgerBlue, "MyPlot");
    }
}

protected override void OnBarUpdate()
{
    // Calculate indicator value
    double value = SMA(20)[0];

    // Assign value to the plot
    Value[0] = value; // or Values[0][0] = value;
}
```

Issue: Strategy Not Executing Orders

Symptoms: - Strategy compiles successfully but doesn't execute any orders - No error messages are shown

Possible Causes: - Entry/exit conditions never met - Insufficient bars for calculation - Account connection issues - Order quantity not specified

Solutions:

```
protected override void OnStateChange()
{
    if (State == State.SetDefaults)
```

```
    {
        Name = "MyStrategy";
        Description = "My custom strategy";

        // Set default quantity
        DefaultQuantity = 1;
    }
}

protected override void OnBarUpdate()
{
    // Skip if not enough bars
    if (CurrentBar < 20)
    {
        Print("Skipping bar " + CurrentBar + " - not enough bars");
        return;
    }

    // Debug entry conditions
    bool condition1 = Close[0] > SMA(20)[0];
    bool condition2 = Close[1] <= SMA(20)[1];
    bool enterLong = condition1 && condition2;

    Print("Bar " + CurrentBar + " - Condition1: " + condition1 + ", Condition2:

    // Enter long if conditions are met
    if (enterLong)
    {
        Print("Entering long at " + Close[0]);
        EnterLong();
    }
}

protected override void OnOrderUpdate(Order order, double limitPrice, double stc
{
    // Log all order updates
    Print("Order Update: " + order.Name + " - State: " + orderState + " - Error:
}
```

Issue: Performance Problems

Symptoms: - Slow execution - High CPU usage - Memory usage grows over time

Possible Causes: - Inefficient calculations - Memory leaks - Excessive logging - Unnecessary calculations on each bar

Solutions:

```
// Cache indicator references
```

```
private SMA sma;

protected override void OnStateChange()
{
    if (State == State.Configure)
    {
        // Create indicator once
        sma = SMA(20);
    }
}

protected override void OnBarUpdate()
{
    // Skip if not enough bars
    if (CurrentBar < 20)
        return;

    // Use cached indicator
    double smaValue = sma[0]; // Instead of SMA(20)[0]

    // Conditional calculations
    if (IsFirstTickOfBar)
    {
        // Perform calculations only once per bar
        // ...
    }

    // Limit logging
    if (CurrentBar % 100 == 0)
    {
        Print("Processing bar " + CurrentBar);
    }
}
```

Best Practices

Code Organization

```
// Well-organized indicator
public class WellOrganizedIndicator : Indicator
{
    // 1. Declare private fields
    private SMA fastSMA;
    private SMA slowSMA;

    // 2. Declare public properties with attributes
    [NinjaScriptProperty]
    [Range(1, 200)]
    [Display(Name = "Fast Period", Description = "Period for the fast SMA", Orde
```

```
    public int FastPeriod { get; set; }

    [NinjaScriptProperty]
    [Range(1, 200)]
    [Display(Name = "Slow Period", Description = "Period for the slow SMA", Orde
    public int SlowPeriod { get; set; }

    // 3. Override OnStateChange for initialization
    protected override void OnStateChange()
    {
        if (State == State.SetDefaults)
        {
            // Set default values
            Name = "Well Organized Indicator";
            Description = "An example of a well-organized indicator";

            // Set default property values
            FastPeriod = 10;
            SlowPeriod = 20;

            // Set indicator properties
            IsOverlay = false;
            Calculate = Calculate.OnBarClose;
        }
        else if (State == State.Configure)
        {
            // Add plots
            AddPlot(Brushes.DodgerBlue, "FastSMA");
            AddPlot(Brushes.Red, "SlowSMA");
            AddPlot(Brushes.Green, "Difference");

            // Initialize indicators
            fastSMA = SMA(FastPeriod);
            slowSMA = SMA(SlowPeriod);
        }
    }

    // 4. Override OnBarUpdate for calculations
    protected override void OnBarUpdate()
    {
        // Skip if not enough bars
        if (CurrentBar < SlowPeriod)
            return;

        // Calculate values
        double fastValue = fastSMA[0];
        double slowValue = slowSMA[0];
        double difference = fastValue - slowValue;

        // Assign values to plots
        Values[0][0] = fastValue;
```

```
        Values[1][0] = slowValue;
        Values[2][0] = difference;
    }

    // 5. Helper methods
    private bool IsCrossing(int barsAgo)
    {
        return (Values[0][barsAgo] > Values[1][barsAgo] && Values[0][barsAgo + 1
            || (Values[0][barsAgo] < Values[1][barsAgo] && Values[0][barsAgo + 1
    }
}
```

Error Handling

```
// Robust error handling
protected override void OnBarUpdate()
{
    try
    {
        // Skip if not enough bars
        if (CurrentBar < 20)
            return;

        // Perform calculations
        double value = CalculateValue();

        // Assign value to plot
        Value[0] = value;
    }
    catch (Exception ex)
    {
        // Log the error
        Print("Error in OnBarUpdate: " + ex.Message);

        // Set a default value
        Value[0] = double.NaN;
    }
}

private double CalculateValue()
{
    try
    {
        // Perform calculation that might fail
        double result = SomeCalculation();

        // Validate result
        if (double.IsNaN(result) || double.IsInfinity(result))
        {
```

```
            throw new InvalidOperationException("Invalid calculation result");
        }

        return result;
    }
    catch (Exception ex)
    {
        // Log the specific error
        Print("Error in CalculateValue: " + ex.Message);

        // Re-throw to be handled by the caller
        throw;
    }
}
```

Documentation

```
/// <summary>
/// A custom indicator that calculates the difference between two moving average
/// </summary>
public class WellDocumentedIndicator : Indicator
{
    private SMA fastSMA;
    private SMA slowSMA;

    /// <summary>
    /// Period for the fast moving average.
    /// </summary>
    [NinjaScriptProperty]
    [Range(1, 200)]
    [Display(Name = "Fast Period", Description = "Period for the fast SMA", Orde
    public int FastPeriod { get; set; }

    /// <summary>
    /// Period for the slow moving average.
    /// </summary>
    [NinjaScriptProperty]
    [Range(1, 200)]
    [Display(Name = "Slow Period", Description = "Period for the slow SMA", Orde
    public int SlowPeriod { get; set; }

    /// <summary>
    /// Initializes the indicator.
    /// </summary>
    protected override void OnStateChange()
    {
        if (State == State.SetDefaults)
        {
            // Set default values
```

```
            Name = "Well Documented Indicator";
            Description = "An example of a well-documented indicator";

            // Set default property values
            FastPeriod = 10;
            SlowPeriod = 20;

            // Set indicator properties
            IsOverlay = false;
            Calculate = Calculate.OnBarClose;
        }
        else if (State == State.Configure)
        {
            // Add plots
            AddPlot(Brushes.DodgerBlue, "FastSMA");
            AddPlot(Brushes.Red, "SlowSMA");
            AddPlot(Brushes.Green, "Difference");

            // Initialize indicators
            fastSMA = SMA(FastPeriod);
            slowSMA = SMA(SlowPeriod);
        }
    }

    /// <summary>
    /// Calculates the indicator values for each bar.
    /// </summary>
    protected override void OnBarUpdate()
    {
        // Skip if not enough bars
        if (CurrentBar < SlowPeriod)
            return;

        // Calculate values
        double fastValue = fastSMA[0];
        double slowValue = slowSMA[0];
        double difference = fastValue - slowValue;

        // Assign values to plots
        Values[0][0] = fastValue;
        Values[1][0] = slowValue;
        Values[2][0] = difference;
    }

    /// <summary>
    /// Determines if the fast and slow moving averages are crossing at the spec
    /// </summary>
    /// <param name="barsAgo">The number of bars ago to check.</param>
    /// <returns>True if the moving averages are crossing, false otherwise.</ret
    private bool IsCrossing(int barsAgo)
    {
```

```
        return (Values[0][barsAgo] > Values[1][barsAgo] && Values[0][barsAgo + 1
            || (Values[0][barsAgo] < Values[1][barsAgo] && Values[0][barsAgo + 1
    }
}
```

This comprehensive guide covers the NinjaTrader API and advanced programming topics for NinjaScript development, providing traders and developers with the knowledge and tools to create sophisticated trading applications.

Machine Learning Integration

This section covers the integration of machine learning models with NinjaScript, including frameworks, implementation approaches, and practical examples.

Machine Learning Frameworks

NinjaScript can integrate with various machine learning frameworks through .NET:

TensorFlow.NET

TensorFlow.NET is a .NET binding for TensorFlow, allowing you to use TensorFlow models in C#:

```
// Add reference to TensorFlow.NET
// Install via NuGet: TensorFlow.NET, TensorFlow.Keras, SciSharp.TensorFlow.Redi

using Tensorflow;
using Tensorflow.Keras;
using Tensorflow.Keras.Engine;
using static Tensorflow.Binding;

// Load a pre-trained TensorFlow model
public class TensorFlowIndicator : Indicator
{
    private Model model;
    private bool modelLoaded;

    protected override void OnStateChange()
    {
        if (State == State.SetDefaults)
        {
            Name = "TensorFlow Indicator";
            Description = "Uses a pre-trained TensorFlow model";
            ModelPath = @"C:\Path\To\Model";
        }
        else if (State == State.DataLoaded)
        {
            try
```

```
            {
                // Load the model
                model = tf.keras.models.load_model(ModelPath);
                modelLoaded = true;
                Print("TensorFlow model loaded successfully");
            }
            catch (Exception ex)
            {
                Print("Error loading TensorFlow model: " + ex.Message);
                modelLoaded = false;
            }
        }
    }

    protected override void OnBarUpdate()
    {
        if (!modelLoaded || CurrentBar < LookbackPeriod)
            return;

        // Prepare input data
        float[] inputData = new float[LookbackPeriod];
        for (int i = 0; i < LookbackPeriod; i++)
        {
            inputData[i] = (float)Close[i];
        }

        // Normalize data
        float mean = inputData.Average();
        float stdDev = (float)Math.Sqrt(inputData.Select(x => Math.Pow(x - mean,
        for (int i = 0; i < inputData.Length; i++)
        {
            inputData[i] = (inputData[i] - mean) / stdDev;
        }

        // Reshape for model input
        var input = tf.reshape(tf.constant(inputData), new int[] { 1, LookbackPe

        // Make prediction
        var output = model.predict(input);
        float prediction = output.numpy().GetValue<float>(0, 0);

        // Denormalize prediction
        prediction = prediction * stdDev + mean;

        // Set indicator value
        Value[0] = prediction;
    }

    [NinjaScriptProperty]
    [Display(Name = "Model Path", Description = "Path to the TensorFlow model",
    public string ModelPath { get; set; }
```

```
    [NinjaScriptProperty]
    [Range(5, 200)]
    [Display(Name = "Lookback Period", Description = "Number of bars to use for
    public int LookbackPeriod { get; set; }
}
```

ML.NET

ML.NET is Microsoft's machine learning framework for .NET:

```
// Add reference to ML.NET
// Install via NuGet: Microsoft.ML

using Microsoft.ML;
using Microsoft.ML.Data;

// Define input and output data classes
public class ModelInput
{
    [LoadColumn(0)]
    public float Open { get; set; }

    [LoadColumn(1)]
    public float High { get; set; }

    [LoadColumn(2)]
    public float Low { get; set; }

    [LoadColumn(3)]
    public float Close { get; set; }

    [LoadColumn(4)]
    public float Volume { get; set; }
}

public class ModelOutput
{
    [ColumnName("Score")]
    public float Prediction { get; set; }
}

// ML.NET Indicator
public class MLNetIndicator : Indicator
{
    private PredictionEngine<ModelInput, ModelOutput> predictionEngine;
    private bool modelLoaded;

    protected override void OnStateChange()
```

```
{
    if (State == State.SetDefaults)
    {
        Name = "ML.NET Indicator";
        Description = "Uses a pre-trained ML.NET model";
        ModelPath = @"C:\Path\To\Model.zip";
    }
    else if (State == State.DataLoaded)
    {
        try
        {
            // Load the model
            MLContext mlContext = new MLContext();
            ITransformer mlModel = mlContext.Model.Load(ModelPath, out var m
            predictionEngine = mlContext.Model.CreatePredictionEngine<ModelI
            modelLoaded = true;
            Print("ML.NET model loaded successfully");
        }
        catch (Exception ex)
        {
            Print("Error loading ML.NET model: " + ex.Message);
            modelLoaded = false;
        }
    }
}

protected override void OnBarUpdate()
{
    if (!modelLoaded)
        return;

    // Prepare input data
    ModelInput input = new ModelInput
    {
        Open = (float)Open[0],
        High = (float)High[0],
        Low = (float)Low[0],
        Close = (float)Close[0],
        Volume = (float)Volume[0]
    };

    // Make prediction
    ModelOutput output = predictionEngine.Predict(input);

    // Set indicator value
    Value[0] = output.Prediction;
}

[NinjaScriptProperty]
[Display(Name = "Model Path", Description = "Path to the ML.NET model", Orde
public string ModelPath { get; set; }
```

```
}
```

Accord.NET

Accord.NET is a machine learning framework for .NET:

```
// Add reference to Accord.NET
// Install via NuGet: Accord.MachineLearning, Accord.Math, Accord.Statistics

using Accord.MachineLearning;
using Accord.Math;
using Accord.Statistics.Models.Regression;
using Accord.Statistics.Models.Regression.Linear;

// Accord.NET Indicator
public class AccordNetIndicator : Indicator
{
    private MultipleLinearRegression regression;
    private bool modelTrained;

    protected override void OnStateChange()
    {
        if (State == State.SetDefaults)
        {
            Name = "Accord.NET Indicator";
            Description = "Uses Accord.NET for linear regression";
            LookbackPeriod = 20;
        }
    }

    protected override void OnBarUpdate()
    {
        if (CurrentBar < LookbackPeriod)
            return;

        // Train the model every N bars or when not trained
        if (!modelTrained || CurrentBar % RetrainInterval == 0)
        {
            TrainModel();
        }

        // Prepare input data for prediction
        double[] input = new double[LookbackPeriod];
        for (int i = 0; i < LookbackPeriod; i++)
        {
            input[i] = Close[i + 1]; // Use previous bars for prediction
        }

        // Make prediction
```

```
            double prediction = regression.Compute(input);

            // Set indicator value
            Value[0] = prediction;
        }

        private void TrainModel()
        {
            // Prepare training data
            double[][] inputs = new double[TrainingBars][];
            double[] outputs = new double[TrainingBars];

            for (int i = 0; i < TrainingBars; i++)
            {
                inputs[i] = new double[LookbackPeriod];
                for (int j = 0; j < LookbackPeriod; j++)
                {
                    inputs[i][j] = Close[i + j + 1];
                }
                outputs[i] = Close[i];
            }

            // Create and train the regression model
            regression = new MultipleLinearRegression(LookbackPeriod);
            regression.Regress(inputs, outputs);

            modelTrained = true;
        }

        [NinjaScriptProperty]
        [Range(5, 200)]
        [Display(Name = "Lookback Period", Description = "Number of bars to use for
        public int LookbackPeriod { get; set; }

        [NinjaScriptProperty]
        [Range(10, 1000)]
        [Display(Name = "Training Bars", Description = "Number of bars to use for tr
        public int TrainingBars { get; set; }

        [NinjaScriptProperty]
        [Range(1, 100)]
        [Display(Name = "Retrain Interval", Description = "Interval in bars for retr
        public int RetrainInterval { get; set; }
    }
```

Implementation Approaches

Pre-trained Models

Using pre-trained models is the simplest approach:

1. Train your model outside of NinjaTrader using Python or another environment
2. Save the model to a file format that can be loaded in C#
3. Load the model in your NinjaScript code
4. Use the model to make predictions

Online Learning

Online learning allows your model to adapt to new data:

```
public class OnlineLearningIndicator : Indicator
{
    private List<double[]> features;
    private List<double> targets;
    private LinearRegression model;

    protected override void OnStateChange()
    {
        if (State == State.SetDefaults)
        {
            Name = "Online Learning Indicator";
            Description = "Uses online learning for prediction";
            LookbackPeriod = 10;
            LearningRate = 0.01;
        }
        else if (State == State.DataLoaded)
        {
            features = new List<double[]>();
            targets = new List<double>();
            model = new LinearRegression(LookbackPeriod);
        }
    }

    protected override void OnBarUpdate()
    {
        if (CurrentBar < LookbackPeriod)
            return;

        // Extract features
        double[] feature = new double[LookbackPeriod];
        for (int i = 0; i < LookbackPeriod; i++)
        {
            feature[i] = Close[i + 1];
        }

        // Make prediction
        double prediction = model.Predict(feature);
        Value[0] = prediction;
```

```
        // Update model with actual value (online learning)
        if (CurrentBar > LookbackPeriod)
        {
            features.Add(feature);
            targets.Add(Close[1]); // Previous bar's close

            // Limit the number of samples to prevent memory issues
            if (features.Count > MaxSamples)
            {
                features.RemoveAt(0);
                targets.RemoveAt(0);
            }

            // Retrain the model
            model.Update(features.ToArray(), targets.ToArray(), LearningRate);
        }
    }

    [NinjaScriptProperty]
    [Range(5, 100)]
    [Display(Name = "Lookback Period", Description = "Number of bars to use for
    public int LookbackPeriod { get; set; }

    [NinjaScriptProperty]
    [Range(0.001, 0.1, 0.001)]
    [Display(Name = "Learning Rate", Description = "Learning rate for model upda
    public double LearningRate { get; set; }

    [NinjaScriptProperty]
    [Range(100, 10000)]
    [Display(Name = "Max Samples", Description = "Maximum number of samples to s
    public int MaxSamples { get; set; }

    // Simple linear regression implementation
    private class LinearRegression
    {
        private double[] weights;
        private double bias;

        public LinearRegression(int features)
        {
            weights = new double[features];
            bias = 0;
        }

        public double Predict(double[] x)
        {
            double sum = bias;
            for (int i = 0; i < weights.Length; i++)
            {
                sum += weights[i] * x[i];
```

```
            }
            return sum;
        }

        public void Update(double[][] x, double[] y, double learningRate)
        {
            int n = x.Length;

            for (int i = 0; i < n; i++)
            {
                double prediction = Predict(x[i]);
                double error = y[i] - prediction;

                // Update bias
                bias += learningRate * error;

                // Update weights
                for (int j = 0; j < weights.Length; j++)
                {
                    weights[j] += learningRate * error * x[i][j];
                }
            }
        }
    }
}
```

Feature Engineering

Effective feature engineering is crucial for ML model performance:

```
public class FeatureEngineeringIndicator : Indicator
{
    private MLModel model;

    protected override void OnStateChange()
    {
        if (State == State.SetDefaults)
        {
            Name = "Feature Engineering Indicator";
            Description = "Uses engineered features for prediction";
            LookbackPeriod = 20;
        }
        else if (State == State.DataLoaded)
        {
            model = new MLModel();
        }
    }

    protected override void OnBarUpdate()
```

```
    {
        if (CurrentBar < LookbackPeriod)
            return;

        // Engineer features
        double[] features = EngineerFeatures();

        // Make prediction
        double prediction = model.Predict(features);

        // Set indicator value
        Value[0] = prediction;
    }

    private double[] EngineerFeatures()
    {
        List<double> features = new List<double>();

        // Price-based features
        features.Add(Close[0] / Open[0] - 1); // Current bar return
        features.Add(Close[1] / Open[1] - 1); // Previous bar return

        // Moving averages
        features.Add(SMA(10)[0] / SMA(20)[0] - 1); // MA ratio

        // Volatility features
        features.Add(ATR(14)[0] / Close[0]); // Normalized ATR

        // Momentum features
        features.Add(ROC(10)[0]); // Rate of change
        features.Add(RSI(14)[0] / 100.0); // Normalized RSI

        // Volume features
        features.Add(Volume[0] / SMA(Volume, 20)[0]); // Volume ratio

        // Time-based features
        DateTime time = Time[0];
        features.Add(Math.Sin(2 * Math.PI * time.Hour / 24)); // Hour of day (si
        features.Add(Math.Cos(2 * Math.PI * time.Hour / 24)); // Hour of day (co
        features.Add(Math.Sin(2 * Math.PI * time.DayOfWeek / 7)); // Day of week
        features.Add(Math.Cos(2 * Math.PI * time.DayOfWeek / 7)); // Day of week

        return features.ToArray();
    }

    // Simplified ML model class
    private class MLModel
    {
        public double Predict(double[] features)
        {
            // Placeholder for actual model prediction
```

```
            // In a real implementation, this would use a trained model
            return features.Average();
        }
    }
}
```

Common Challenges and Solutions

Memory Management

Machine learning models can consume significant memory:

```
// Proper resource disposal
protected override void OnStateChange()
{
    if (State == State.Terminated)
    {
        // Dispose model resources
        if (model != null)
        {
            model.Dispose();
            model = null;
        }

        // Force garbage collection
        GC.Collect();
    }
}
```

Performance Optimization

Optimize performance for real-time trading:

```
// Cache predictions to avoid recalculating
private Dictionary<int, double> predictionCache = new Dictionary<int, double>();

protected override void OnBarUpdate()
{
    // Check if prediction is already cached
    if (predictionCache.ContainsKey(CurrentBar))
    {
        Value[0] = predictionCache[CurrentBar];
        return;
    }

    // Calculate prediction
```

```
    double prediction = CalculatePrediction();

    // Cache the prediction
    predictionCache[CurrentBar] = prediction;

    // Set indicator value
    Value[0] = prediction;

    // Limit cache size
    if (predictionCache.Count > 1000)
    {
        int oldestKey = predictionCache.Keys.Min();
        predictionCache.Remove(oldestKey);
    }
}
```

Error Handling

Robust error handling is essential:

```
protected override void OnBarUpdate()
{
    try
    {
        // Model prediction code
        double prediction = model.Predict(features);
        Value[0] = prediction;
    }
    catch (Exception ex)
    {
        // Log error
        Print("Error in ML model prediction: " + ex.Message);

        // Fallback to a simple prediction
        Value[0] = SMA(20)[0];

        // Attempt to recover
        if (!modelRecoveryAttempted)
        {
            try
            {
                // Reload or reinitialize the model
                InitializeModel();
                modelRecoveryAttempted = true;
            }
            catch (Exception recoveryEx)
            {
                Print("Model recovery failed: " + recoveryEx.Message);
            }
```

```
        }
    }
}
```

Practical Examples

Price Direction Predictor

```
public class PriceDirectionPredictor : Indicator
{
    private LogisticRegression model;
    private List<double[]> features;
    private List<bool> targets;

    protected override void OnStateChange()
    {
        if (State == State.SetDefaults)
        {
            Name = "Price Direction Predictor";
            Description = "Predicts price direction using logistic regression";
            LookbackPeriod = 10;
            TrainingPeriod = 100;
        }
        else if (State == State.DataLoaded)
        {
            features = new List<double[]>();
            targets = new List<bool>();
            model = new LogisticRegression(LookbackPeriod * 3); // 3 features pe
        }
    }

    protected override void OnBarUpdate()
    {
        if (CurrentBar < LookbackPeriod + 1)
            return;

        // Extract features
        double[] feature = ExtractFeatures();

        // Train model
        if (CurrentBar >= TrainingPeriod && CurrentBar % 10 == 0)
        {
            TrainModel();
        }

        // Make prediction if model is trained
        if (features.Count >= TrainingPeriod)
        {
```

```
            double probability = model.Predict(feature);
            Value[0] = probability;

            // Draw arrows based on prediction
            if (probability > 0.7)
                Draw.ArrowUp(this, "Up" + CurrentBar, false, 0, Low[0] - TickSiz
            else if (probability < 0.3)
                Draw.ArrowDown(this, "Down" + CurrentBar, false, 0, High[0] + Ti
        }

        // Add current bar to training data
        if (CurrentBar > 0)
        {
            features.Add(feature);
            targets.Add(Close[0] > Close[1]); // Target: did price go up?

            // Limit training data size
            if (features.Count > MaxTrainingSize)
            {
                features.RemoveAt(0);
                targets.RemoveAt(0);
            }
        }
    }

    private double[] ExtractFeatures()
    {
        List<double> featureList = new List<double>();

        // For each bar in the lookback period
        for (int i = 1; i <= LookbackPeriod; i++)
        {
            // Price change
            featureList.Add(Close[i] / Close[i + 1] - 1);

            // Normalized range
            featureList.Add((High[i] - Low[i]) / Close[i]);

            // Volume change
            featureList.Add(Volume[i] / Volume[i + 1] - 1);
        }

        return featureList.ToArray();
    }

    private void TrainModel()
    {
        if (features.Count < TrainingPeriod)
            return;

        double[][] trainingFeatures = features.ToArray();
```

```
        bool[] trainingTargets = targets.ToArray();

        model.Train(trainingFeatures, trainingTargets);
    }

    [NinjaScriptProperty]
    [Range(5, 50)]
    [Display(Name = "Lookback Period", Description = "Number of bars to use for
    public int LookbackPeriod { get; set; }

    [NinjaScriptProperty]
    [Range(50, 500)]
    [Display(Name = "Training Period", Description = "Number of bars to use for
    public int TrainingPeriod { get; set; }

    [NinjaScriptProperty]
    [Range(100, 5000)]
    [Display(Name = "Max Training Size", Description = "Maximum number of sample
    public int MaxTrainingSize { get; set; }

    // Simple logistic regression implementation
    private class LogisticRegression
    {
        private double[] weights;
        private double bias;
        private double learningRate = 0.01;
        private int maxIterations = 100;

        public LogisticRegression(int features)
        {
            weights = new double[features];
            bias = 0;
        }

        public double Predict(double[] x)
        {
            double z = bias;
            for (int i = 0; i < weights.Length; i++)
            {
                z += weights[i] * x[i];
            }
            return Sigmoid(z);
        }

        public void Train(double[][] x, bool[] y)
        {
            int n = x.Length;
            double[] yDouble = Array.ConvertAll(y, item => item ? 1.0 : 0.0);

            for (int iter = 0; iter < maxIterations; iter++)
            {
```

```
                double[] gradWeights = new double[weights.Length];
                double gradBias = 0;

                for (int i = 0; i < n; i++)
                {
                    double prediction = Predict(x[i]);
                    double error = yDouble[i] - prediction;

                    gradBias += error;
                    for (int j = 0; j < weights.Length; j++)
                    {
                        gradWeights[j] += error * x[i][j];
                    }
                }

                bias += learningRate * gradBias / n;
                for (int j = 0; j < weights.Length; j++)
                {
                    weights[j] += learningRate * gradWeights[j] / n;
                }
            }
        }

        private double Sigmoid(double z)
        {
            return 1.0 / (1.0 + Math.Exp(-z));
        }
    }
}
```

By understanding these concepts and examples, you can effectively integrate machine learning models with NinjaScript to create advanced trading indicators and strategies.

Converting from PineScript to NinjaScript

This section provides guidance on converting trading indicators and strategies from PineScript (used in TradingView) to NinjaScript (used in NinjaTrader), including syntax comparisons, conversion patterns, and practical examples.

Language Differences

PineScript and NinjaScript have fundamental differences in their syntax and programming paradigms:

Syntax Comparison

Concept	PineScript	NinjaScript

Language Base	Custom scripting language	C# (object-oriented)
Variable Declaration	`var x = 10`	`int x = 10;`
Function Definition	`f(x) => x * 2`	`private double F(double x) { return x * 2; }`
Comments	`// Comment`	`// Comment` or `/* Multi-line comment */`
Arrays	`array.new_float(size, initial_value)`	`double[] array = new double[size];`
Conditional	`if (condition) x else y`	`condition ? x : y` or `if (condition) { x; } else { y; }`
Loops	`for i = 0 to 10`	`for (int i = 0; i < 10; i++)`

Data Access

Concept	PineScript	NinjaScript
Current Price	`close`	`Close[0]`
Historical Price	`close[1]`	`Close[1]`
Volume	`volume`	`Volume[0]`
Time	`time`	`Time[0]`
Bar Index	`bar_index`	`CurrentBar`

Indicator Creation

Concept	PineScript	NinjaScript
Indicator Declaration	`indicator(title="My Indicator", overlay=true)`	`public class MyIndicator : Indicator { ... }`
Plot	`plot(series, title="Plot", color=color.blue)`	`AddPlot(Brushes.Blue, "Plot");` and `Value[0] = series;`
Color Condition	`plot(series, color=close > open ? color.green : color.red)`	`PlotBrushes[0][0] = Close[0] > Open[0] ? Brushes.Green : Brushes.Red;`
Transparency	`color.new(color.blue, 70)`	`Brushes.Blue.Clone();`

		`brush.Opacity = 0.3;`

Conversion Patterns

Basic Indicator Structure

PineScript:

```
//@version=5
indicator("Simple MA", overlay=true)
len = input.int(14, title="Length")
src = input(close, title="Source")
ma = ta.sma(src, len)
plot(ma, color=color.blue, linewidth=2)
```

NinjaScript:

```
using System;
using System.ComponentModel;
using NinjaTrader.Cbi;
using NinjaTrader.Data;
using NinjaTrader.NinjaScript;

namespace NinjaTrader.NinjaScript.Indicators
{
    public class SimpleMA : Indicator
    {
        protected override void OnStateChange()
        {
            if (State == State.SetDefaults)
            {
                Name = "Simple MA";
                Description = "Simple Moving Average";
                IsOverlay = true;

                // Default parameters
                Length = 14;
                Source = PriceType.Close;
            }
            else if (State == State.Configure)
            {
                // Add plot
                AddPlot(new Stroke(Brushes.Blue, 2), PlotStyle.Line, "MA");
            }
        }

        protected override void OnBarUpdate()
        {
```

```
            // Skip calculation until we have enough bars
            if (CurrentBar < Length)
                return;

            // Get source price
            double src = GetPrice(Source);

            // Calculate MA
            double ma = SMA(src, Length)[0];

            // Set indicator value
            Value[0] = ma;
        }

        private double GetPrice(PriceType priceType)
        {
            switch (priceType)
            {
                case PriceType.Close: return Close[0];
                case PriceType.Open: return Open[0];
                case PriceType.High: return High[0];
                case PriceType.Low: return Low[0];
                case PriceType.Median: return (High[0] + Low[0]) / 2;
                case PriceType.Typical: return (High[0] + Low[0] + Close[0]) / 3
                case PriceType.Weighted: return (High[0] + Low[0] + Close[0] + C
                default: return Close[0];
            }
        }

        [NinjaScriptProperty]
        [Range(1, 200)]
        [Display(Name = "Length", Description = "The period for the moving avera
        public int Length { get; set; }

        [NinjaScriptProperty]
        [Display(Name = "Source", Description = "The price source for the moving
        public PriceType Source { get; set; }

        public enum PriceType
        {
            Close,
            Open,
            High,
            Low,
            Median,
            Typical,
            Weighted
        }
    }
}
```

Converting Built-in Functions

Many PineScript built-in functions have equivalents in NinjaScript:

PineScript	NinjaScript
`ta.sma(src, len)`	`SMA(src, len)[0]`
`ta.ema(src, len)`	`EMA(src, len)[0]`
`ta.rsi(src, len)`	`RSI(src, len)[0]`
`ta.macd(src, fast, slow, signal)`	`MACD(fast, slow, signal).Diff[0]`
`ta.crossover(a, b)`	`CrossAbove(a, b, 1)`
`ta.crossunder(a, b)`	`CrossBelow(a, b, 1)`
`ta.highest(src, len)`	`MAX(src, len)[0]`
`ta.lowest(src, len)`	`MIN(src, len)[0]`
`ta.stoch(src, high, low, len)`	`Stochastic(len, 1, 1).K[0]`
`ta.atr(len)`	`ATR(len)[0]`

Converting Strategy Logic

PineScript:

```
//@version=5
strategy("Simple Crossover Strategy", overlay=true)

// Inputs
fastLength = input.int(10, title="Fast Length")
slowLength = input.int(20, title="Slow Length")

// Calculate MAs
fastMA = ta.sma(close, fastLength)
slowMA = ta.sma(close, slowLength)

// Plot MAs
plot(fastMA, color=color.blue, linewidth=2, title="Fast MA")
plot(slowMA, color=color.red, linewidth=2, title="Slow MA")

// Entry conditions
longCondition = ta.crossover(fastMA, slowMA)
shortCondition = ta.crossunder(fastMA, slowMA)

// Execute trades
```

```
if (longCondition)
    strategy.entry("Long", strategy.long)

if (shortCondition)
    strategy.entry("Short", strategy.short)
```

NinjaScript:

```
using System;
using System.ComponentModel;
using NinjaTrader.Cbi;
using NinjaTrader.Data;
using NinjaTrader.NinjaScript;

namespace NinjaTrader.NinjaScript.Strategies
{
    public class SimpleCrossoverStrategy : Strategy
    {
        private SMA fastMA;
        private SMA slowMA;

        protected override void OnStateChange()
        {
            if (State == State.SetDefaults)
            {
                Name = "Simple Crossover Strategy";
                Description = "Enters trades based on SMA crossovers";

                // Default parameters
                FastLength = 10;
                SlowLength = 20;
            }
            else if (State == State.Configure)
            {
                // Initialize indicators
                fastMA = SMA(FastLength);
                slowMA = SMA(SlowLength);

                // Add plots
                AddPlot(new Stroke(Brushes.Blue, 2), PlotStyle.Line, "Fast MA");
                AddPlot(new Stroke(Brushes.Red, 2), PlotStyle.Line, "Slow MA");
            }
        }

        protected override void OnBarUpdate()
        {
            // Skip calculation until we have enough bars
            if (CurrentBar < SlowLength)
                return;
```

```
            // Update plot values
            Values[0][0] = fastMA[0];
            Values[1][0] = slowMA[0];

            // Entry conditions
            bool longCondition = CrossAbove(fastMA, slowMA, 1);
            bool shortCondition = CrossBelow(fastMA, slowMA, 1);

            // Execute trades
            if (longCondition)
                EnterLong();

            if (shortCondition)
                EnterShort();
        }

        [NinjaScriptProperty]
        [Range(1, 100)]
        [Display(Name = "Fast Length", Description = "The period for the fast mo
        public int FastLength { get; set; }

        [NinjaScriptProperty]
        [Range(1, 100)]
        [Display(Name = "Slow Length", Description = "The period for the slow mo
        public int SlowLength { get; set; }
    }
}
```

Common Challenges and Solutions

Series vs. Values

PineScript uses series for all values, while NinjaScript distinguishes between series and scalar values:

PineScript:

```
myValue = close > open ? 1 : -1
```

NinjaScript:

```
double myValue = Close[0] > Open[0] ? 1 : -1;
```

Historical Data Access

PineScript uses negative indices for future data and positive indices for historical data, while

NinjaScript uses positive indices for historical data:

PineScript:

```
prevClose = close[1]  // Previous bar's close
```

NinjaScript:

```
double prevClose = Close[1];  // Previous bar's close
```

Custom Functions

PineScript uses simple function definitions, while NinjaScript uses C# methods:

PineScript:

```
calcAverage(a, b) => (a + b) / 2
myAvg = calcAverage(high, low)
```

NinjaScript:

```
private double CalcAverage(double a, double b)
{
    return (a + b) / 2;
}

// Usage
double myAvg = CalcAverage(High[0], Low[0]);
```

Color Handling

PineScript has simple color handling, while NinjaScript uses the .NET Brush system:

PineScript:

```
bullColor = color.green
bearColor = color.red
barColor = close > open ? bullColor : bearColor
plot(close, color=barColor)
```

NinjaScript:

```
protected override void OnBarUpdate()
{
    // Set plot color based on bar type
    PlotBrushes[0][0] = Close[0] > Open[0] ? Brushes.Green : Brushes.Red;
```

```
    // Set indicator value
    Value[0] = Close[0];
}
```

Alert Handling

PineScript has built-in alert functions, while NinjaScript uses a different approach:

PineScript:

```
if (longCondition)
    alert("Long Signal", alert.freq_once_per_bar)
```

NinjaScript:

```
if (longCondition)
{
    if (AlertsEnabled)
    {
        Alert("MyAlertName", Priority.High, "Long Signal",
              NinjaTrader.Core.Globals.InstallDir + @"\sounds\Alert1.wav",
              10, Brushes.Green, Brushes.White);
    }
}
```

Practical Examples

MACD Indicator Conversion

PineScript:

```
//@version=5
indicator("MACD", overlay=false)

// Input parameters
fastLength = input.int(12, title="Fast Length")
slowLength = input.int(26, title="Slow Length")
signalLength = input.int(9, title="Signal Length")
src = input(close, title="Source")

// Calculate MACD
[macdLine, signalLine, histLine] = ta.macd(src, fastLength, slowLength, signalLe

// Plot
plot(macdLine, color=color.blue, title="MACD Line")
```

```
plot(signalLine, color=color.red, title="Signal Line")
plot(histLine, color=histLine > 0 ? color.green : color.red, style=plot.style_hi
```

NinjaScript:

```
using System;
using System.ComponentModel;
using System.Windows.Media;
using NinjaTrader.Cbi;
using NinjaTrader.Data;
using NinjaTrader.NinjaScript;

namespace NinjaTrader.NinjaScript.Indicators
{
    public class CustomMACD : Indicator
    {
        private EMA fastEMA;
        private EMA slowEMA;
        private EMA signalEMA;

        protected override void OnStateChange()
        {
            if (State == State.SetDefaults)
            {
                Name = "Custom MACD";
                Description = "Moving Average Convergence Divergence";
                IsOverlay = false;

                // Default parameters
                FastLength = 12;
                SlowLength = 26;
                SignalLength = 9;
                Source = PriceType.Close;
            }
            else if (State == State.Configure)
            {
                // Add plots
                AddPlot(new Stroke(Brushes.Blue, 2), PlotStyle.Line, "MACD Line"
                AddPlot(new Stroke(Brushes.Red, 2), PlotStyle.Line, "Signal Line
                AddPlot(new Stroke(Brushes.Green, 2), PlotStyle.Bar, "Histogram"

                // Initialize indicators
                fastEMA = EMA(FastLength);
                slowEMA = EMA(SlowLength);
                signalEMA = EMA(SignalLength);
            }
        }

        protected override void OnBarUpdate()
```

```
        {
            // Skip calculation until we have enough bars
            if (CurrentBar < SlowLength)
                return;

            // Get source price
            double src = GetPrice(Source);

            // Calculate MACD components
            double macdLine = fastEMA[0] - slowEMA[0];

            // Store MACD line for signal calculation
            Values[0][0] = macdLine;

            // Calculate signal line using stored MACD line values
            if (IsFirstTickOfBar)
                signalEMA.Input[0] = macdLine;

            double signalLine = signalEMA[0];
            Values[1][0] = signalLine;

            // Calculate histogram
            double histLine = macdLine - signalLine;
            Values[2][0] = histLine;

            // Set histogram color
            PlotBrushes[2][0] = histLine > 0 ? Brushes.Green : Brushes.Red;
        }

        private double GetPrice(PriceType priceType)
        {
            switch (priceType)
            {
                case PriceType.Close: return Close[0];
                case PriceType.Open: return Open[0];
                case PriceType.High: return High[0];
                case PriceType.Low: return Low[0];
                case PriceType.Median: return (High[0] + Low[0]) / 2;
                case PriceType.Typical: return (High[0] + Low[0] + Close[0]) / 3
                case PriceType.Weighted: return (High[0] + Low[0] + Close[0] + C
                default: return Close[0];
            }
        }

        [NinjaScriptProperty]
        [Range(1, 100)]
        [Display(Name = "Fast Length", Description = "The period for the fast EM
        public int FastLength { get; set; }

        [NinjaScriptProperty]
        [Range(1, 100)]
```

```
        [Display(Name = "Slow Length", Description = "The period for the slow EM
        public int SlowLength { get; set; }

        [NinjaScriptProperty]
        [Range(1, 100)]
        [Display(Name = "Signal Length", Description = "The period for the signa
        public int SignalLength { get; set; }

        [NinjaScriptProperty]
        [Display(Name = "Source", Description = "The price source for calculatio
        public PriceType Source { get; set; }

        public enum PriceType
        {
            Close,
            Open,
            High,
            Low,
            Median,
            Typical,
            Weighted
        }
    }
}
```

RSI Strategy Conversion

PineScript:

```
//@version=5
strategy("RSI Strategy", overlay=false)

// Input parameters
length = input.int(14, title="RSI Length")
overbought = input.int(70, title="Overbought Level")
oversold = input.int(30, title="Oversold Level")

// Calculate RSI
rsiValue = ta.rsi(close, length)

// Plot RSI
plot(rsiValue, color=color.blue, title="RSI")
hline(overbought, color=color.red, linestyle=hline.style_dashed)
hline(oversold, color=color.green, linestyle=hline.style_dashed)

// Entry conditions
longCondition = ta.crossover(rsiValue, oversold)
shortCondition = ta.crossunder(rsiValue, overbought)
```

```
// Execute trades
if (longCondition)
    strategy.entry("Long", strategy.long)

if (shortCondition)
    strategy.entry("Short", strategy.short)
```

NinjaScript:

```
using System;
using System.ComponentModel;
using System.Windows.Media;
using NinjaTrader.Cbi;
using NinjaTrader.Data;
using NinjaTrader.NinjaScript;

namespace NinjaTrader.NinjaScript.Strategies
{
    public class RSIStrategy : Strategy
    {
        private RSI rsiIndicator;

        protected override void OnStateChange()
        {
            if (State == State.SetDefaults)
            {
                Name = "RSI Strategy";
                Description = "Enters trades based on RSI levels";

                // Default parameters
                Length = 14;
                Overbought = 70;
                Oversold = 30;
            }
            else if (State == State.Configure)
            {
                // Initialize indicators
                rsiIndicator = RSI(Length, 1);

                // Add plot
                AddPlot(new Stroke(Brushes.Blue, 2), PlotStyle.Line, "RSI");

                // Add horizontal lines
                AddHorizontalLine(Overbought, Brushes.Red, DashStyleHelper.Dash,
                AddHorizontalLine(Oversold, Brushes.Green, DashStyleHelper.Dash,
            }
        }

        protected override void OnBarUpdate()
        {
```

```
            // Skip calculation until we have enough bars
            if (CurrentBar < Length)
                return;

            // Update plot value
            Values[0][0] = rsiIndicator[0];

            // Entry conditions
            bool longCondition = CrossAbove(rsiIndicator, Oversold, 1);
            bool shortCondition = CrossBelow(rsiIndicator, Overbought, 1);

            // Execute trades
            if (longCondition)
                EnterLong();

            if (shortCondition)
                EnterShort();
        }

        [NinjaScriptProperty]
        [Range(2, 100)]
        [Display(Name = "RSI Length", Description = "The period for the RSI calc
        public int Length { get; set; }

        [NinjaScriptProperty]
        [Range(50, 100)]
        [Display(Name = "Overbought Level", Description = "The overbought thresh
        public int Overbought { get; set; }

        [NinjaScriptProperty]
        [Range(0, 50)]
        [Display(Name = "Oversold Level", Description = "The oversold threshold"
        public int Oversold { get; set; }
    }
}
```

By understanding these conversion patterns and examples, you can effectively translate your PineScript indicators and strategies to NinjaScript, leveraging the power and flexibility of the NinjaTrader platform.

Common Errors and Solutions

This section covers common errors encountered when developing with NinjaScript and their solutions, providing a comprehensive troubleshooting guide.

Compilation Errors

CS0103: The name 'X' does not exist in the current context

Error Message:

```
CS0103: The name 'X' does not exist in the current context
```

Causes: - Missing using directive for the namespace containing the symbol - Typo in the variable or method name - Attempting to use a variable outside its scope - Referencing a class or method that doesn't exist

Solutions: 1. Add the appropriate using directive at the top of your file `csharp using NinjaTrader.Indicator; using NinjaTrader.Data; using System.Collections.Generic;` 2. Check for typos in variable or method names 3. Ensure the variable is declared in the current scope 4. Verify the class or method exists in the referenced assemblies

CS0246: The type or namespace name 'X' could not be found

Error Message:

```
CS0246: The type or namespace name 'X' could not be found (are you missing a usi
```

Causes: - Missing using directive - Missing assembly reference - Typo in the type or namespace name

Solutions: 1. Add the appropriate using directive 2. Add the required assembly reference 3. Check for typos in the type or namespace name

CS0122: 'X' is inaccessible due to its protection level

Error Message:

```
CS0122: 'X' is inaccessible due to its protection level
```

Causes: - Attempting to access a private or protected member from outside its scope - Trying to use an internal class from another assembly

Solutions: 1. Use only public members of classes you don't own 2. Change the access modifier of your own classes/members if appropriate 3. Use the appropriate accessor methods if available

CS1061: 'X' does not contain a definition for 'Y'

Error Message:

```
CS1061: 'X' does not contain a definition for 'Y' and no extension method 'Y' ac
```

Causes: - Typo in the property or method name - Using a property or method that doesn't exist on the object - Using a property or method from a newer version of NinjaTrader

Solutions: 1. Check for typos in the property or method name 2. Verify the property or method exists on the object 3. Check the NinjaTrader version compatibility

CS0234: The type or namespace name 'X' does not exist in the namespace 'Y'

Error Message:

```
CS0234: The type or namespace name 'X' does not exist in the namespace 'Y' (are
```

Causes: - Missing assembly reference - Incorrect namespace - Typo in the namespace or type name

Solutions: 1. Add the required assembly reference 2. Check for typos in the namespace or type name 3. Verify the correct namespace for the type

Runtime Errors

NullReferenceException

Error Message:

```
System.NullReferenceException: Object reference not set to an instance of an obj
```

Causes: - Using an object that hasn't been initialized - Accessing a property or method on a null object - Using an indicator or data series before it's ready

Solutions: 1. Initialize objects before using them 2. Add null checks before accessing properties or methods `csharp if (myObject != null) { myObject.DoSomething(); }` 3. Ensure indicators and data series are ready before using them `csharp if (CurrentBar < 20 || myIndicator == null) return;`

IndexOutOfRangeException

Error Message:

```
System.IndexOutOfRangeException: Index was outside the bounds of the array
```

Causes: - Accessing an array element with an index that's negative or beyond the array's length - Using an incorrect index when accessing price or indicator data

Solutions: 1. Check array bounds before accessing elements `csharp if (index >= 0 &&`

`index < array.Length) { var value = array[index]; }` 2. Ensure you have enough bars before accessing historical data `csharp if (CurrentBar < period) return;`

InvalidOperationException

Error Message:

```
System.InvalidOperationException: Operation is not valid due to the current stat
```

Causes: - Performing an operation that's not allowed in the current state - Calling methods in the wrong order - Modifying a collection while iterating through it

Solutions: 1. Check the object's state before performing operations 2. Follow the correct sequence of method calls 3. Use a temporary collection for modifications during iteration

ArgumentException

Error Message:

```
System.ArgumentException: Value does not fall within the expected range
```

Causes: - Passing an invalid argument to a method - Using an invalid parameter value

Solutions: 1. Validate arguments before passing them to methods 2. Check the documentation for valid parameter ranges 3. Use appropriate error handling for boundary cases

NinjaScript-Specific Errors

"Indicator is still calculating"

Error Message:

```
Indicator is still calculating and has not completed its historical data process
```

Causes: - Attempting to access indicator values before historical calculation is complete - Using an indicator in a strategy before it's ready

Solutions: 1. Check the `IsFirstTickOfBar` property before accessing indicator values 2. Use the `Calculate` property to control when calculations occur 3. Add appropriate checks in your code `csharp if (BarsInProgress != 0 || CurrentBars[0] < 20) return;`

"Cannot call method during historical data processing"

Error Message:

```
Cannot call method during historical data processing
```

Causes: - Calling certain methods that are only allowed during real-time processing - Attempting to place orders during historical processing in a strategy

Solutions: 1. Check the `IsHistorical` property before calling such methods `csharp if (!IsHistorical) { // Call methods only allowed in real-time }` 2. Use the `OnMarketData()` method for real-time processing

"Series index out of range"

Error Message:

```
Series index out of range
```

Causes: - Accessing a data point that doesn't exist yet - Using an incorrect index when accessing price or indicator data

Solutions: 1. Check that you have enough bars before accessing data `csharp if (CurrentBar < period) return;` 2. Use proper indexing (remember that index 0 is the current bar)

"BarsInProgress out of range"

Error Message:

```
BarsInProgress out of range
```

Causes: - Accessing a data series that doesn't exist - Using an incorrect BarsInProgress index

Solutions: 1. Ensure you've added all required data series 2. Check BarsInProgress before accessing data `csharp if (BarsInProgress >= DataSeriesCount) return;`

Strategy-Specific Errors

"Order rejected: Insufficient funds"

Error Message:

```
Order rejected: Insufficient funds
```

Causes: - Attempting to place an order that requires more funds than available - Incorrect position size calculation

Solutions: 1. Implement proper position sizing based on account size 2. Add checks for available funds before placing orders 3. Use the `Account.Cash` property to determine available funds

"Order rejected: Outside of regular trading hours"

Error Message:

```
Order rejected: Outside of regular trading hours
```

Causes: - Attempting to place an order outside of the instrument's trading hours - Not accounting for market sessions

Solutions: 1. Check the `Time[0]` against the instrument's session hours 2. Use the `Bars.IsFirstBarOfSession` property to detect session boundaries 3. Implement session filters in your strategy

"Strategy is not enabled for live trading"

Error Message:

```
Strategy is not enabled for live trading
```

Causes: - Attempting to use a strategy in live trading without enabling it - Missing required configuration for live trading

Solutions: 1. Enable the strategy for live trading in the strategy properties 2. Complete all required configuration steps for live trading 3. Ensure the strategy has been properly tested in simulation

Performance Issues

Slow Calculation

Symptoms: - Indicator or strategy calculations take a long time - UI becomes unresponsive during calculations - High CPU usage

Causes: - Inefficient algorithms - Redundant calculations - Excessive logging or drawing - Too many indicators or data series

Solutions: 1. Optimize algorithms and avoid redundant calculations 2. Cache results instead of recalculating ```csharp private Dictionary calculationCache = new Dictionary();

protected override void OnBarUpdate() { if (calculationCache.ContainsKey(CurrentBar)) { Value[0] = calculationCache[CurrentBar]; return; }

```
    // Perform calculation
    double result = PerformCalculation();

    // Cache the result
    calculationCache[CurrentBar] = result;
```

```
    Value[0] = result;
```

} ``` 3. Limit logging and drawing operations 4. Reduce the number of indicators and data series

Memory Leaks

Symptoms: - Increasing memory usage over time - NinjaTrader becomes slower the longer it runs - Eventually crashes due to out of memory errors

Causes: - Not disposing of resources properly - Accumulating large collections without bounds - Creating new objects in frequently called methods

Solutions: 1. Properly dispose of resources that implement IDisposable `csharp protected override void OnStateChange() { if (State == State.Terminated) { // Dispose resources if (myResource != null) { myResource.Dispose(); myResource = null; } } }` 2. Limit the size of collections `csharp if (myCollection.Count > maxSize) { myCollection.RemoveAt(0); }` 3. Reuse objects instead of creating new ones 4. Use value types (structs) for small, frequently created objects

Debugging Techniques

Using Print Statements

Print statements are the simplest way to debug NinjaScript code:

```
Print("Debug: CurrentBar = " + CurrentBar + ", Close = " + Close[0]);
```

For more control over output, use different trace levels:

```
if (TraceLevel >= TraceLevel.Verbose)
    Print("Verbose debug info: " + detailedInfo);
```

Visual Debugging

Use drawing objects to visualize values and conditions on charts:

```
// Draw a dot at a specific point
Draw.Dot(this, "Debug" + CurrentBar, false, 0, Value[0], Brushes.Yellow);

// Draw text with a value
Draw.Text(this, "DebugText" + CurrentBar, "Value: " + Math.Round(Value[0], 2), 0
```

Using Try-Catch Blocks

Wrap potentially problematic code in try-catch blocks:

```
try
{
    // Code that might throw an exception
    double result = CalculateValue();
    Value[0] = result;
}
catch (Exception ex)
{
    Print("Error in calculation: " + ex.Message);
    // Handle the error gracefully
    Value[0] = double.NaN;
}
```

Logging to File

For persistent logging, write to a file:

```
private StreamWriter logFile;

protected override void OnStateChange()
{
    if (State == State.SetDefaults)
    {
        // Standard setup
    }
    else if (State == State.Configure)
    {
        // Open log file
        string logPath = @"C:\Logs\MyIndicator_" + DateTime.Now.ToString("yyyyMM
        logFile = new StreamWriter(logPath, true);
    }
    else if (State == State.Terminated)
    {
        // Close log file
        if (logFile != null)
        {
            logFile.Close();
            logFile.Dispose();
            logFile = null;
        }
    }
}

private void Log(string message)
{
    if (logFile != null)
    {
```

```
        string timestamp = DateTime.Now.ToString("yyyy-MM-dd HH:mm:ss.fff");
        logFile.WriteLine(timestamp + " - " + message);
        logFile.Flush();
    }
}
```

By understanding these common errors and their solutions, you can more effectively troubleshoot issues in your NinjaScript development and create more robust indicators and strategies.

Complete List of NinjaTrader Indicators and Settings

This comprehensive reference documents all built-in indicators in NinjaTrader, their parameters, and usage examples.

Table of Contents

- Introduction to NinjaTrader Indicators
- Indicator Categories
- Complete Indicator Reference
- Custom Indicator Development

Introduction to NinjaTrader Indicators

NinjaTrader comes with a rich set of built-in technical indicators that can be applied to charts and used in strategies. These indicators cover various analysis techniques including trend analysis, momentum, volatility, volume analysis, and more.

Using Indicators in NinjaTrader

Indicators can be added to charts through: - Chart right-click menu → Indicators - Strategy Builder - NinjaScript code

Common Indicator Properties

All indicators share these common properties: - **Calculate**: Determines when calculations occur (OnBarClose, OnPriceChange, OnEachTick) - **DisplayInDataBox**: Controls visibility in the data box - **IsOverlay**: Determines if the indicator is displayed on the price panel or separate panel - **ScaleJustification**: Controls the scale placement (Right, Left, Overlay) - **Displacement**: Shifts the indicator forward/backward by specified bars - **BarsRequiredToPlot**: Minimum number of bars required before plotting

Indicator Categories

Trend Indicators

- Moving Averages (SMA, EMA, WMA, etc.)
- MACD
- ADX/DMI
- Parabolic SAR
- Ichimoku Cloud

Momentum Indicators

- RSI
- Stochastics
- CCI
- Williams %R
- Momentum

Volume Indicators

- Volume
- On Balance Volume
- Volume Profile
- Money Flow Index
- Accumulation/Distribution

Volatility Indicators

- Bollinger Bands
- Average True Range
- Standard Deviation
- Keltner Channel
- Donchian Channel

Oscillators

- MACD
- RSI
- Stochastics
- CCI
- Ultimate Oscillator

Complete Indicator Reference

Accumulation/Distribution (ADL)

Description: Measures the cumulative flow of money into and out of a security using price and volume.

Parameters: - None (uses price and volume data from the chart)

Example Usage:

```
// Adding ADL indicator to a chart
ADL adl = ADL();
```

Adaptive Price Zone (APZ)

Description: Creates a volatility-based envelope around price.

Parameters: - Period (default: 21) - Bandwidth (default: 2)

Example Usage:

```
// Adding APZ with custom parameters
APZ apz = APZ(14, 3);
```

Aroon

Description: Measures the time between highs and lows over a time period.

Parameters: - Period (default: 14)

Example Usage:

```
// Adding Aroon indicator
Aroon aroon = Aroon(14);
```

Aroon Oscillator

Description: Calculates the difference between Aroon Up and Aroon Down.

Parameters: - Period (default: 14)

Example Usage:

```
// Adding Aroon Oscillator
AroonOscillator aroonOsc = AroonOscillator(14);
```

Average Directional Index (ADX)

Description: Measures the strength of a trend regardless of direction.

Parameters: - Period (default: 14)

Example Usage:

```
// Adding ADX indicator
ADX adx = ADX(14);
```

Average Directional Movement Rating (ADXR)

Description: Averages the current ADX with the ADX from a previous period.

Parameters: - Period (default: 14)

Example Usage:

```
// Adding ADXR indicator
ADXR adxr = ADXR(14);
```

Average True Range (ATR)

Description: Measures market volatility by decomposing the entire range of an asset price for a period.

Parameters: - Period (default: 14)

Example Usage:

```
// Adding ATR indicator
ATR atr = ATR(14);
```

Balance of Power (BOP)

Description: Measures the strength of buyers vs. sellers by assessing the ability to push price to an extreme.

Parameters: - None (uses OHLC data from the chart)

Example Usage:

```
// Adding BOP indicator
BOP bop = BOP();
```

Block Volume

Description: Displays volume as blocks based on a specified size threshold.

Parameters: - Block Size (default: 1000)

Example Usage:

```
// Adding Block Volume indicator
BlockVolume blockVol = BlockVolume(5000);
```

Bollinger Bands

Description: Creates a price envelope based on standard deviations from a moving average.

Parameters: - Period (default: 20) - Standard Deviation (default: 2) - MA Type (default: Simple)

Example Usage:

```
// Adding Bollinger Bands
Bollinger bollinger = Bollinger(20, 2);
```

BuySell Pressure

Description: Measures buying and selling pressure based on price movement and volume.

Parameters: - Smooth Period (default: 3)

Example Usage:

```
// Adding BuySell Pressure indicator
BuySellPressure buySellPressure = BuySellPressure(3);
```

BuySell Volume

Description: Separates volume into buying and selling components.

Parameters: - None (uses price and volume data from the chart)

Example Usage:

```
// Adding BuySell Volume indicator
BuySellVolume buySellVolume = BuySellVolume();
```

Camarilla Pivots

Description: Calculates support and resistance levels based on the previous period's range.

Parameters: - Pivot Range (default: Daily)

Example Usage:

```
// Adding Camarilla Pivots
```

```
CamarillaPivots camarilla = CamarillaPivots(PivotRange.Daily);
```

CandleStickPattern

Description: Detects specific candlestick patterns.

Parameters: - Pattern (ChartPattern enum) - Trend Strength (default: 4)

Supported Patterns: - BearishBeltHold - BearishEngulfing - BearishHarami - BearishHaramiCross - BullishBeltHold - BullishEngulfing - BullishHarami - BullishHaramiCross - DarkCloudCover - Doji - DownsideTasukiGap - EveningStar - FallingThreeMethods - Hammer - HangingMan - InvertedHammer - MorningStar - PiercingLine - RisingThreeMethods - ShootingStar - ThreeBlackCrows - ThreeWhiteSoldiers - UpsideTasukiGap

Example Usage:

```
// Detecting a bullish engulfing pattern
CandleStickPattern engulfing = CandleStickPattern(ChartPattern.BullishEngulfing,

// Using in a strategy
if (engulfing[0] == 1)
    EnterLong();
```

Chaikin Money Flow

Description: Measures the amount of Money Flow Volume over a specific period.

Parameters: - Period (default: 20)

Example Usage:

```
// Adding Chaikin Money Flow
ChaikinMoneyFlow cmf = ChaikinMoneyFlow(20);
```

Chaikin Oscillator

Description: Measures the momentum of the Accumulation/Distribution Line.

Parameters: - Fast Period (default: 3) - Slow Period (default: 10)

Example Usage:

```
// Adding Chaikin Oscillator
ChaikinOscillator chaikinOsc = ChaikinOscillator(3, 10);
```

Chaikin Volatility

Description: Measures the rate of change of the Average True Range.

Parameters: - Period (default: 14) - ROC Period (default: 10) - MA Type (default: EMA)

Example Usage:

```
// Adding Chaikin Volatility
ChaikinVolatility chaikinVol = ChaikinVolatility(14, 10);
```

Chande Momentum Oscillator (CMO)

Description: Calculates the difference between the sum of recent gains and the sum of recent losses.

Parameters: - Period (default: 14)

Example Usage:

```
// Adding CMO indicator
CMO cmo = CMO(14);
```

Commodity Channel Index (CCI)

Description: Identifies cyclical trends in price movement.

Parameters: - Period (default: 14)

Example Usage:

```
// Adding CCI indicator
CCI cci = CCI(14);
```

Correlation

Description: Measures the statistical correlation between two data series.

Parameters: - Period (default: 14) - Input Series 1 - Input Series 2

Example Usage:

```
// Correlation between two instruments
Correlation corr = Correlation(14, Close, SMA(20));
```

Darvas Boxes

Description: Identifies trading ranges based on recent highs and lows.

Parameters: - Box Percent (default: 4) - Noise Percent (default: 1.5)

Example Usage:

```
// Adding Darvas Boxes
Darvas darvas = Darvas(4, 1.5);
```

Directional Movement (DM)

Description: Measures the directional movement of price.

Parameters: - Period (default: 14)

Example Usage:

```
// Adding DM indicator
DM dm = DM(14);
```

Directional Movement Index (DMI)

Description: Combines DI+ and DI- to show the direction of the trend.

Parameters: - Period (default: 14)

Example Usage:

```
// Adding DMI indicator
DMI dmi = DMI(14);
```

Donchian Channel

Description: Plots the highest high and lowest low over a specified period.

Parameters: - Period (default: 20)

Example Usage:

```
// Adding Donchian Channel
DonchianChannel donchian = DonchianChannel(20);
```

Double Stochastics

Description: Applies the stochastic formula twice to smooth the indicator.

Parameters: - Period (default: 14) - K Period (default: 3) - D Period (default: 3)

Example Usage:

```
// Adding Double Stochastics
DoubleStochastics doubleStoch = DoubleStochastics(14, 3, 3);
```

Dynamic Momentum Index (DMIndex)

Description: A variable-length RSI that adjusts based on volatility.

Parameters: - Period (default: 14) - Standard Deviation (default: 1.5) - Moving Average Type (default: SMA)

Example Usage:

```
// Adding Dynamic Momentum Index
DMIndex dmIndex = DMIndex(14, 1.5);
```

Ease of Movement

Description: Relates price change to volume, emphasizing days when price changes with low volume.

Parameters: - Period (default: 14)

Example Usage:

```
// Adding Ease of Movement
EaseOfMovement eom = EaseOfMovement(14);
```

Fisher Transform

Description: Converts prices into a Gaussian normal distribution.

Parameters: - Period (default: 10)

Example Usage:

```
// Adding Fisher Transform
FisherTransform fisher = FisherTransform(10);
```

Forecast Oscillator (FOSC)

Description: Compares the current price with the Time Series Forecast.

Parameters: - Period (default: 14)

Example Usage:

```
// Adding Forecast Oscillator
FOSC fosc = FOSC(14);
```

Keltner Channel

Description: Creates a volatility-based envelope using ATR around an EMA.

Parameters: - Period (default: 20) - ATR Period (default: 10) - Multiplier (default: 2)

Example Usage:

```
// Adding Keltner Channel
KeltnerChannel keltner = KeltnerChannel(20, 10, 2);
```

Linear Regression

Description: Plots a straight line through price data using linear regression.

Parameters: - Period (default: 14)

Example Usage:

```
// Adding Linear Regression
LinearRegression linReg = LinearRegression(14);
```

MACD

Description: Moving Average Convergence/Divergence indicator that shows the relationship between two moving averages.

Parameters: - Fast Period (default: 12) - Slow Period (default: 26) - Signal Period (default: 9)

Example Usage:

```
// Adding MACD indicator
MACD macd = MACD(12, 26, 9);
```

Market Facilitation Index

Description: Evaluates market efficiency by comparing price range to volume.

Parameters: - None (uses price and volume data from the chart)

Example Usage:

```
// Adding Market Facilitation Index
```

```
MarketFacilitationIndex mfi = MarketFacilitationIndex();
```

Momentum

Description: Measures the amount that price has changed over a given period.

Parameters: - Period (default: 14)

Example Usage:

```
// Adding Momentum indicator
Momentum momentum = Momentum(14);
```

Money Flow Index (MFI)

Description: A volume-weighted RSI that measures money flow in and out of a security.

Parameters: - Period (default: 14)

Example Usage:

```
// Adding Money Flow Index
MoneyFlowIndex mfi = MoneyFlowIndex(14);
```

Moving Average - Exponential (EMA)

Description: Calculates an exponentially weighted moving average giving more weight to recent prices.

Parameters: - Period (default: 14)

Example Usage:

```
// Adding EMA indicator
EMA ema = EMA(14);
```

Moving Average - Simple (SMA)

Description: Calculates the unweighted mean of the previous n periods.

Parameters: - Period (default: 14)

Example Usage:

```
// Adding SMA indicator
SMA sma = SMA(14);
```

Moving Average - Triangular (TMA)

Description: A double-smoothed simple moving average.

Parameters: - Period (default: 14)

Example Usage:

```
// Adding TMA indicator
TMA tma = TMA(14);
```

Moving Average - Weighted (WMA)

Description: Assigns a weighting factor to each value in the period, with the most recent having the highest weight.

Parameters: - Period (default: 14)

Example Usage:

```
// Adding WMA indicator
WMA wma = WMA(14);
```

Moving Average - Hull (HMA)

Description: A faster and smoother moving average that reduces lag.

Parameters: - Period (default: 14)

Example Usage:

```
// Adding HMA indicator
HMA hma = HMA(14);
```

Moving Average - Variable (VMA)

Description: A moving average that adjusts its period based on volatility.

Parameters: - Period (default: 14) - Volatility (default: 2)

Example Usage:

```
// Adding VMA indicator
VMA vma = VMA(14, 2);
```

On Balance Volume (OBV)

Description: Relates price changes to volume, accumulating volume on up days and subtracting it on down days.

Parameters: - None (uses price and volume data from the chart)

Example Usage:

```
// Adding OBV indicator
OBV obv = OBV();
```

Parabolic SAR

Description: Provides potential entry and exit points with dots placed below or above price.

Parameters: - Acceleration Factor (default: 0.02) - Acceleration Limit (default: 0.2)

Example Usage:

```
// Adding Parabolic SAR
ParabolicSAR psar = ParabolicSAR(0.02, 0.2);
```

Pivot Points

Description: Calculates support and resistance levels based on previous period's high, low, and close.

Parameters: - Pivot Range (default: Daily)

Example Usage:

```
// Adding Pivot Points
PivotPoints pivots = PivotPoints(PivotRange.Daily);
```

Price Oscillator

Description: Shows the difference between two moving averages as a percentage.

Parameters: - Fast Period (default: 12) - Slow Period (default: 26) - Smooth Period (default: 9)

Example Usage:

```
// Adding Price Oscillator
PriceOscillator priceOsc = PriceOscillator(12, 26, 9);
```

Rate of Change (ROC)

Description: Calculates the percentage change between the current price and the price n

periods ago.

Parameters: - Period (default: 14)

Example Usage:

```
// Adding ROC indicator
ROC roc = ROC(14);
```

Relative Strength Index (RSI)

Description: Measures the speed and change of price movements on a scale from 0 to 100.

Parameters: - Period (default: 14) - Smooth Period (default: 3)

Example Usage:

```
// Adding RSI indicator
RSI rsi = RSI(14, 3);
```

Standard Deviation

Description: Measures the dispersion of a set of values from their average.

Parameters: - Period (default: 14)

Example Usage:

```
// Adding Standard Deviation
StdDev stdDev = StdDev(14);
```

Stochastics

Description: Compares a security's closing price to its price range over a given period.

Parameters: - Period (default: 14) - K Period (default: 3) - D Period (default: 3) - Stochastic Type (default: Fast)

Example Usage:

```
// Adding Stochastics indicator
Stochastics stoch = Stochastics(14, 3, 3);
```

Stochastics Fast

Description: A faster version of the stochastic oscillator.

Parameters: - Period (default: 14) - K Period (default: 3) - D Period (default: 3)

Example Usage:

```
// Adding Fast Stochastics
StochasticsFast stochFast = StochasticsFast(14, 3, 3);
```

Stochastics RSI

Description: Applies the stochastic formula to RSI values.

Parameters: - Period (default: 14) - K Period (default: 3) - D Period (default: 3)

Example Usage:

```
// Adding Stochastics RSI
StochasticsRSI stochRSI = StochasticsRSI(14, 3, 3);
```

Swing

Description: Identifies swing highs and lows in price movement.

Parameters: - Strength (default: 4)

Example Usage:

```
// Adding Swing indicator
Swing swing = Swing(4);
```

Time Series Forecast (TSF)

Description: Projects the price n bars into the future using linear regression.

Parameters: - Period (default: 14)

Example Usage:

```
// Adding Time Series Forecast
TSF tsf = TSF(14);
```

Ultimate Oscillator

Description: Uses multiple timeframes to avoid the pitfalls of using a single timeframe.

Parameters: - Fast Period (default: 7) - Intermediate Period (default: 14) - Slow Period (default: 28)

Example Usage:

```
// Adding Ultimate Oscillator
UltimateOscillator uo = UltimateOscillator(7, 14, 28);
```

Volume

Description: Displays trading volume.

Parameters: - None (uses volume data from the chart)

Example Usage:

```
// Adding Volume indicator
Volume vol = Volume();
```

Volume Moving Average

Description: Calculates a moving average of volume.

Parameters: - Period (default: 14) - MA Type (default: SMA)

Example Usage:

```
// Adding Volume Moving Average
VolumeMA volMA = VolumeMA(14);
```

Volume Oscillator

Description: Shows the difference between two volume moving averages.

Parameters: - Fast Period (default: 12) - Slow Period (default: 26)

Example Usage:

```
// Adding Volume Oscillator
VolumeOscillator volOsc = VolumeOscillator(12, 26);
```

Volume Profile

Description: Shows the volume traded at each price level.

Parameters: - Period Type (default: Session) - Profile Type (default: VOC) - Value Area Percent (default: 70)

Example Usage:

```
// Adding Volume Profile
VolumeProfile volProfile = VolumeProfile();
```

Volume Rate of Change

Description: Calculates the percentage change in volume over a specified period.

Parameters: - Period (default: 14)

Example Usage:

```
// Adding Volume Rate of Change
VolumeROC volROC = VolumeROC(14);
```

Volume Up Down

Description: Separates volume into up and down components based on price movement.

Parameters: - None (uses price and volume data from the chart)

Example Usage:

```
// Adding Volume Up Down
VolumeUpDown volUpDown = VolumeUpDown();
```

Williams %R

Description: A momentum indicator that measures overbought/oversold levels.

Parameters: - Period (default: 14)

Example Usage:

```
// Adding Williams %R
WilliamsR williamsR = WilliamsR(14);
```

Woodies CCI

Description: A modified version of the CCI indicator with trend lines.

Parameters: - CCI Period (default: 14) - Long Cycle Period (default: 34)

Example Usage:

```
// Adding Woodies CCI
WoodiesCCI woodiesCCI = WoodiesCCI(14, 34);
```

ZigZag

Description: Identifies significant price reversals.

Parameters: - Deviation (default: 5) - Type (default: Percent)

Example Usage:

```
// Adding ZigZag indicator
ZigZag zigZag = ZigZag(5);
```

Custom Indicator Development

NinjaTrader allows you to create custom indicators using NinjaScript. Here's a basic template:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.ComponentModel.DataAnnotations;
using System.Windows.Media;
using NinjaTrader.Gui;
using NinjaTrader.Gui.Chart;
using NinjaTrader.NinjaScript;

namespace NinjaTrader.NinjaScript.Indicators
{
    public class MyCustomIndicator : Indicator
    {
        protected override void OnStateChange()
        {
            if (State == State.SetDefaults)
            {
                Description = "My custom indicator description";
                Name = "MyCustomIndicator";

                // Default properties
                Period = 14;

                // Plotting properties
                AddPlot(new Stroke(Brushes.DodgerBlue, 2), PlotStyle.Line, "MyPl
            }
        }

        protected override void OnBarUpdate()
        {
            // Skip calculation until we have enough bars
            if (CurrentBar < Period)
                return;
```

```
            // Indicator calculation logic
            double sum = 0;
            for (int i = 0; i < Period; i++)
                sum += Close[i];

            // Set the plot value
            Value[0] = sum / Period;
        }

        [NinjaScriptProperty]
        [Range(1, int.MaxValue)]
        [Display(Name = "Period", Description = "Number of bars to calculate", C
        public int Period
        { get; set; }
    }
}
```

For more advanced indicator development, refer to the NinjaScript Indicators Development section of this knowledge base.