EasyLanguage Optimization API Developer's Guide

January 8, 2016

IMPORTANT INFORMATION:

No offer or solicitation to buy or sell securities, securities derivative, futures products or offexchange foreign currency (forex) transactions of any kind, or any type of trading or investment advice, recommendation or strategy, is made, given or in any manner endorsed by any TradeStation affiliate and the information made available on this Website is not an offer or solicitation of any kind in any jurisdiction where any TradeStation affiliate is not authorized to do business, including but not limited to Japan.

Past performance, whether actual or indicated by historical tests of strategies, is no guarantee of future performance or success. There is a possibility that you may sustain a loss equal to or greater than your entire investment regardless of which asset class you trade (equities, options futures or forex); therefore, you should not invest or risk money that you cannot afford to lose. Options trading is not suitable for all investors. Your account application to trade options will be considered and approved or disapproved based on all relevant factors, including your trading experience. Please click here to view the document titled Characteristics and Risks of Standardized Options. Before trading any asset class, customers must read the relevant risk disclosure statements on our Other Information page. System access and trade placement and execution may be delayed or fail due to market volatility and volume, quote delays, system and software errors, Internet traffic, outages and other factors.

TradeStation Group, Inc. Affiliates: All proprietary technology in TradeStation is owned by TradeStation Technologies, Inc. Equities, equities options, and commodity futures products and services are offered by TradeStation Securities, Inc. (Member NYSE, FINRA, NFA and SIPC). TradeStation Securities, Inc.'s SIPC coverage is available only for equities and equities options accounts. Forex products and services are offered by TradeStation Forex, a division of IBFX, Inc. (Member NFA).

Table of Contents

EasyLanguage Optimization API

Introduction

The EasyLanguage Optimization API gives you all the tools you need to define and run optimization jobs using EasyLanguage. It gives you control over every aspect of an optimization:

- Define the security (or securities) to use for the optimization, including the symbol, interval, and history range. You can even optimize over multiple symbols or intervals.
- Define the strategies to use for the optimization.
- Specify the inputs to optimize. You can now optimize over a list of values or expressions as well as over a range of values.
- Define the global settings for the optimization, as well as any strategy-specific settings.
- Monitor the progress of the optimization as it runs.
- Retrieve and analyze the optimization results when the optimization is done.
- Handle optimization errors if they occur.

Since optimizations are now fully programmable, you can even write an application that executes multiple optimizations automatically. In fact, you can use the results of one optimization to determine the parameters of the next optimization; this opens up new possibilities for strategy analysis.

The Optimization API also takes full advantage of the multi-threaded optimization engine. This requires no extra effort on your part. Just specify what you want to optimize and how to respond to events, and the optimization will automatically use all the cores on your machine. (You can also specify how many threads to use for the optimization if you wish.)

A Simple Optimization App

Before we delve into the details of the API, here is a simple optimization app written in EasyLanguage. Take a minute to read through the code and get the flavor of the API. The following sections will explore the techniques used by the application in depth.

This TradingApp optimizes the Bollinger Bands LE and SE strategies for the symbol, minute interval, and history range specified in the inputs. The application optimizes the number of deviations for each strategy (NumDevsUp and NumDevsDn). The step size to use for the optimization is also specified as an input.

The application outputs its progress to the Print Log as it runs, and it outputs the best Net Profit when the optimization has completed.

If you type in this code yourself, be sure to assign the `InitApp` method to the TradingApp's `Initialized` event.

Note that this application will automatically run a multi-threaded optimization if you have multiple cores on your machine.

```
using elsystem;
using elsystem.windows.forms;
using elsystem.drawing;

inputs:
   Sym("MSFT"),
   int MinuteInterval(5),
   LastDate("09/30/2013"),
   int DaysBack(90),
   NumDevsStep(0.1);

vars:
   Form form1(null),
   tsopt.Optimizer optimizer(null);

method void InitApp( elsystem.Object sender, elsystem.InitializedEventArgs args )
vars:
   Button startButton;
begin
   form1 = Form.Create("OptTest", 500, 500);
   form1.BackColor = Color.LightGray;
   form1.Dock = DockStyle.Right;

   startButton = Button.Create("Start Optimization", 120, 30);
   startButton.Location(20, 20);
   startButton.Click += OnStartButtonClick;
   form1.AddControl(startButton);

   form1.Show();
end;

method void OnStartButtonClick(elsystem.Object sender, elsystem.EventArgs args)
begin
   StartOptimization();
end;

method tsopt.Job DefineJob()
vars:
   tsopt.Job job,
   tsopt.Security security,
   tsopt.Strategy strategy;
begin
   job = new tsopt.Job;

   security = job.Securities.AddSecurity();
   security.Symbol = Sym;
   security.Interval.SetMinuteChart(MinuteInterval);
   security.History.LastDateString = LastDate;
   security.History.DaysBack = DaysBack;

   strategy = job.Strategies.AddStrategy("Bollinger Bands LE");
   strategy.ELInputs.OptRange("NumDevsDn", 1, 3, NumDevsStep);

   strategy = job.Strategies.AddStrategy("Bollinger Bands SE");
   strategy.ELInputs.OptRange("NumDevsUp", 1, 3, NumDevsStep);

   return job;
end;
```

```
method void StartOptimization()
vars:
   tsopt.Job job;
begin
   ClearPrintLog;

   Print("Starting optimization...");

   job = DefineJob();

   optimizer = new tsopt.Optimizer;

   optimizer.JobDone += OptDone;
   optimizer.JobFailed += OptError;
   optimizer.ProgressChanged += OptProgress;

   optimizer.StartJob(job);
end;

method void OptProgress(Object sender, tsopt.ProgressChangedEventArgs args)
begin
   Print("Test ", args.Progress.TestNum.ToString(), " of ",
      args.Progress.TestCount.ToString());
   Print("   ", args.BestValues.FitnessName, " = ",
      args.BestValues.FitnessValue.ToString());
end;

method void OptDone(object sender, tsopt.JobDoneEventArgs args)
begin
   Print("Optimization done");
   Print("Net Profit = ", args.Results.NetProfit());
end;

method void OptError(Object sender, tsopt.JobFailedEventArgs args)
begin
   Print("Optimization Error: ", args.Error.Message);
end;
```

Optimization API Overview

All of the optimization classes reside within the `tsopt` namespace. Since the namespace identifier is very short and the classes have simple names that might conflict with other identifiers, *importing the namespace is **not** recommended*. When writing code, it is easy and quick just to type `tsopt.` and then select a class from the pop-up list provided by auto-complete. This also makes it easy to find the classes that are part of the Optimization API.

Throughout the code examples in this guide, we will always include the namespace qualifier with the class name. This makes it very clear that the class is related to optimizations. We often name the variable after the class, but without the namespace qualifier. You may find this a useful convention in your code as well. For example, the following code snippet declares a variable called `optimizer` as an instance of the `tsopt.Optimizer` class:

```
   vars:
      tsopt.Optimizer optimizer(null);
```

Here is a quick overview of the primary classes in the Optimization API.

The tsopt.Job Class

The `tsopt.Job` class allows a client application to define an optimization job. This definition includes the securities to be used in the optimization, the strategies to be optimized, and any relevant settings.

A job definition always starts with the `Job` class, but there are also a number of helper classes that describe different parts of the job. For example, a `Job` object contains a `tsopt.Strategies` object that defines the strategies used in an optimization. The `Strategies` object is a collection of `tsopt.Strategy` objects, each of which defines a strategy. A `Strategy` object in turn has a `tsopt.ELInputs` object which describes the strategy inputs, and so on. The structure of a job definition is described in more detail below.

The tsopt.Optimizer Class

The `tsopt.Optimizer` class provides methods to start or cancel an optimization job. This is also where you add the event handlers for the optimization.

The tsopt.ProgressInfo Class

The `tsopt.ProgressInfo` class provides information about the progress of an optimization, such as the current test number, the time elapsed and remaining, and for genetic optimizations, the current generation and individual. You can retrieve a `ProgressInfo` object from the `ProgressChangedEventArgs` argument of the `ProgressChanged` event handler.

The tsopt.BestValues Class

The `tsopt.BestValues` class provides information about the best fitness result seen so far during an optimization, along with the optimized parameters associated with that result. Like `ProgressInfo`, this object can be retrieved from the `ProgressChangedEventArgs` argument of the `ProgressChanged` event handler.

The tsopt.Results Class

The `tsopt.Results` class holds the results of an optimization. It can be retrieved from the `JobDoneEventArgs` argument of the `JobDone` event handler. The `Results` class provides methods to retrieve the strategy metrics for each optimization test in the results set.

The tsopt.OptimizationException Class

The `tsopt.OptimizationException` class is an exception that can be thrown during the construction of a job definition. You can catch this exception in order to debug certain kinds of coding errors in the definition process (e.g., indexing into an empty `Strategies` object). The Optimization API also validates a job definition at the start of an optimization to ensure that it is complete and correct. If there are any validation errors, the library will throw a `tsopt.OptimizationException`.

Note that the Optimization API does not throw exceptions while an optimization is *running*. Instead, the error is reported by calling your `JobFailed` event handler. An optimization executes asynchronously, so this technique works better for reporting errors that occur during an optimization.

Error handling is discussed in more detail below.

Defining an Optimization Job

This section provides an overview of the job definition API. For detailed information about each of the job definition classes, see the *API Reference* later in this document.

Structure of a Job Definition

A job definition is a tree with three main nodes: Securities, Strategies, and Settings.

The **Securities** node must contain one or more Security nodes that define the securities to use in an optimization. Each Security node corresponds to a different data series (i.e., Data1, Data2, Data3, etc. in TradeStation). Many optimizations will use only one Security.

The **Strategies** node must contain one or more Strategy nodes that define the strategies to use in an optimization. The inputs and settings for a specific strategy are specified within its Strategy node.

The **Settings** node can contain one or more nodes that define various global settings for the optimization. It is permissible for the Settings node to be empty, in which case the default settings are used.

A job definition may also specify the **OptimizationMethod** to use (which can be either Exhaustive or Genetic). If the optimization method is not specified, it defaults to Exhaustive.

Every **Security** node must contain information about the symbol, interval, and history range for the security. If any of this information is missing, the optimizer will throw an exception when you start the job.

A **Strategy** node requires only the strategy name. If no other information is provided, the default inputs and settings for the strategy will be used. However, most job definitions will provide additional information about a strategy.

A job definition can specify that certain parameters within a Strategy or Security should be optimized. For example, a common case is to optimize one or more inputs in a strategy. In the Optimization API, this would normally be specified by calling the `OptRange` or `OptList` method for the relevant input. It is possible to optimize some parts of a security as well. For example, you can optimize over a list of symbols with the `OptSymbol` property, or you can optimize over a list of intervals with the `OptInterval` property.

To create a job definition, you must start by creating a `tsopt.Job` object:

```
method tsopt.Job DefineJob()
vars:
   tsopt.Job job;
begin
   job = new tsopt.Job;

   // Define the job here

   return job;
end;
```

After creating the job, you can call properties and methods on the `Job` object to define the different parts of the job. The following sections describe two different approaches for specifying a job: (1) defining a job in the traditional style; or (2) defining a job with the "tree style." You can use either or both of these approaches within a single application.

Note that a job definition is essentially *declarative*. It consists of a series of property and method calls that *declare* the contents of a job, but the job definition does not actually *do* anything on its own. Once you define a job, you must pass the definition to the `tsopt.Optimizer.StartJob` method in order to execute the job.

Since the job definition is declarative, it can easily be translated to and from XML, which is another declarative format.

Defining a Job in the Traditional Style

You can build up a job definition using a traditional style of coding where each statement either creates an object, sets a property on the object, or calls a method on the object.

When you use this style, you will need to define some local variables to hold the various sub-objects that you create. (It's possible to write code in the traditional style without local variables, but it results in code that is very repetitive and hard to read.)

At a minimum, you should define local variables for the `tsopt.Security` and `tsopt.Strategy` objects, since you will always need to add at least one security and strategy to the definition:

```
vars:
   tsopt.Security security,
   tsopt.Strategy strategy;
```

To add a security to the definition, get the `Securities` object from the job and call its `AddSecurity` method, assigning the result to your local variable:

```
security = job.Securities.AddSecurity();
```

You can then define the symbol, interval, and history range for the security.

To add a strategy to the definition, get the `Strategies` object from the job and call its `AddStrategy` method, assigning the result to your local variable:

```
strategy = job.Strategies.AddStrategy("Bollinger Bands LE");
```

You can then define the information for the strategy, such as the inputs to optimize.

You can repeat this pattern for each strategy you wish to add to the definition. It's fine to reuse the same local variable for each strategy you add.

If you want to add settings to your job definition, you may wish to define a local variable for each group of settings. This is not required, but it makes the code less repetitive. For example, if you want to change several genetic options, you can create a local variable for the `GeneticOptions` object:

```
vars:
   tsopt.GeneticOptions geneticOptions;
```

Then you can assign the `GeneticOptions` object from the job definition to this variable, so that you can use the variable to set the properties:

```
geneticOptions = job.Settings.GeneticOptions;
geneticOptions.PopulationSize = 200;
geneticOptions.Generations = 75;
geneticOptions.CrossoverRate = 0.8;
geneticOptions.MutationRate = 0.06;
```

The following example shows a method that defines an optimization job in the traditional style:

```
method tsopt.Job DefineJob()
vars:
   tsopt.Job job,
   tsopt.Security security,
   tsopt.Strategy strategy,
   tsopt.GeneticOptions geneticOptions,
   tsopt.ResultOptions resultOptions;
begin
   job = new tsopt.Job;

   job.OptimizationMethod = tsopt.OptimizationMethod.Genetic;

   security = job.Securities.AddSecurity();
   security.Symbol = "CSCO";
   security.Interval.SetMinuteChart(15);
   security.History.LastDateString = "12/31/2012";
   security.History.DaysBack = 180;

   strategy = job.Strategies.AddStrategy("Bollinger Bands LE");
   strategy.ELInputs.OptRange("Length", 10, 30, 1);
   strategy.ELInputs.OptRange("NumDevsDn", 1, 3, 0.1);

   strategy = job.Strategies.AddStrategy("Bollinger Bands SE");
   strategy.ELInputs.OptRange("Length", 10, 30, 1);
   strategy.ELInputs.OptRange("NumDevsUp", 1, 3, 0.1);

   geneticOptions = job.Settings.GeneticOptions;
   geneticOptions.PopulationSize = 200;
   geneticOptions.Generations = 75;

   resultOptions = job.Settings.ResultOptions;
   resultOptions.FitnessMetric = tsopt.MetricID.TSIndex;
   resultOptions.NumTestsToKeep = 300;

   return job;
end;
```

Defining a Job with the Tree Style

It can be very convenient to define an entire job with a single statement, particularly when prototyping or testing an application. The Optimization API supports this style of job definition because method calls can be chained together. When defining a job this way, it is helpful to use indentation to highlight the tree structure. Thus, we call this the "tree style" of job definition.

This is best illustrated with an example:

```
job
   .SetOptimizationMethod(tsopt.OptimizationMethod.Genetic)
   .Securities
      .AddSecurity()
         .SetSymbol("CSCO")
         .Interval
            .SetMinuteChart(5)
         .EndInterval
         .History
            .SetLastDate("12/31/2012")
            .SetDaysBack(180)
         .EndHistory
      .EndSecurity
   .EndSecurities
   .Strategies
      .AddStrategy("Bollinger Bands LE")
         .ELInputs
            .OptRange("Length", 10, 30, 1)
            .OptRange("NumDevsDn", 1, 3, 0.1)
         .EndELInputs
      .EndStrategy
      .AddStrategy("Bollinger Bands SE")
         .ELInputs
            .OptRange("Length", 10, 30, 1)
            .OptRange("NumDevsUp", 1, 3, 0.1)
         .EndELInputs
      .EndStrategy
   .EndStrategies
   .Settings
      .GeneticOptions
         .SetPopulationSize(200)
         .SetGenerations(75)
      .EndGeneticOptions
      .ResultOptions
         .SetFitnessMetric(tsopt.MetricID.TSIndex)
         .SetNumTestsToKeep(300)
      .EndResultOptions
   .EndSettings
.UseDefinition();
```

The tree style has several useful features:

- Since this style emphasizes the tree structure of a job definition in its syntax and formatting, it can make it easier to see and understand the overall structure of a job.
- There's no need to define any local variables. When you add or access sub-objects in the tree, you can immediately call the relevant methods or properties to define that object. For example, the `AddSecurity` method returns an empty `Security` object. Rather than saving that object in a

variable, you can just call methods or properties directly on the returned object to define the security. To define multiple aspects of the security, just chain the calls together.

- When you are "inside" a particular object, you can use its `End...` property to return to the context of its parent object. For example, you can use the `EndInterval` property of the `Interval` object to return to its parent `Security` object, and you can use the `EndSecurity` property of the `Security` object to return to its parent `Securities` object. This allows you to "navigate" through the tree as you are defining the job.
- The tree style has very little repetition. Once you are "inside" an object, you can just chain methods and properties together to define all of the desired options. There is no need to repeat the receiver object (or chain of objects) on each line of code, as is often required with the traditional style. This is particularly helpful in definitions that optimize many parameters or change many of the default settings.
- Auto-complete works very well with the tree style. Just type the dot operator at the beginning of each line, and auto-complete will pop up a list of the methods and properties that are relevant for the current part of the tree. For example, if you are inside the `History` object, you can see all the methods for setting the history range. Most of the definition methods start with "Set…" (to set a property), "Add…" (to add an object or value), or "Opt…" (to optimize a parameter).

The tree style is completely optional. If you feel more comfortable coding job definitions in the traditional style, that's fine; you will still have access to all the power of the Optimization API. However, if you are familiar with tree-based formats such as XML, you may wish to give the tree style a try. Once you get the hang of it, it can be a very efficient and intuitive way to define an optimization job.

When defining a job with chained methods, it is helpful to use the indenting style from the example above. It is not required for the code to work, but it makes the structure of the job definition easy to see and understand. The TradeStation Development Environment makes it simple to write code with this indenting style, since you can press Tab to move to the next indent level or Shift+Tab to back up to the previous indent level. Also, if you start each line with a dot, auto-complete will pop up the available methods and properties for the current part of the job tree. (It is highly recommended that you enable auto-complete while working with the Optimization API.)

Note that you cannot end a tree-style definition with an `End...` property, since EasyLanguage doesn't allow a statement to end with a property (unless it's part of an assignment). Thus, if the last part of your definition is an `End...` property, you must end the definition with a `UseDefinition()` method call. This turns the tree definition into a valid EasyLanguage statement. If you get a syntax error complaining about "An equals sign '=' expected here," just add the call to `UseDefinition()` and it will fix the error.

For a detailed explanation of how the tree style works, see *Appendix 1: Anatomy of a Tree-Style Definition*. For some useful pointers about working with the tree style, see *Appendix 2: Working with the Tree Style*.

Defining a Job Incrementally

In a UI-based optimization application, the job definition is typically not defined all at once. Instead, it is assembled incrementally from input provided by the user. The Optimization API fully supports this style of job definition. In fact, it is so flexible that you can use the job definition as the sole internal representation of the options selected by the user. There is no need to create separate data structures to

hold the user's optimization choices: you can put them straight into the job definition and update them as needed based on user input.

To make incremental updates to a job, start with the `tsopt.Job` object and call the appropriate methods and/or properties to navigate to the desired part of the job tree. You can chain calls together to keep your code concise. For example, suppose you want to change a 5 minute interval to a 10 minute interval. If your Job object is called "job", you can do it like this:

```
job.Securities[0].Interval.SetMinuteChart(10);
```

For properties like `Securities` that return a collection of objects, you can index into the property to retrieve one of its sub-objects. In the example above, we use `Securities[0]` to retrieve the first Security object in the job definition; then we call `Interval` to get its `Interval` object; and finally we call `SetMinuteChart` to set the interval.

When you first create a `Job` object, it is completely empty except for the `Securities`, `Strategies`, and `Settings` nodes. Thus, you must call the appropriate `Add…` methods to add securities and strategies to the job. (If you tried the example code above on a newly created job, you would get an error, since the `Securities` collection would be empty.) Since every optimization job must contain at least one security, you might want to add a `Security` node immediately after creating the `Job` object:

```
job = new tsopt.Job;
job.Securities.AddSecurity();
```

You might even want to provide a default security that users can modify:

```
job = new tsopt.Job;

security = job.Securities.AddSecurity();
security.Symbol = "CSCO";
security.Interval.SetMinuteChart(5);
security.History.LastDate = DateTime.Today;
security.History.DaysBack = 30;
```

If you need to make multiple changes at a particular spot in the job definition tree, it's a good idea to navigate to that part of the tree and save the object in a local variable. Then you can make repeated calls on that object to apply the changes. This is more efficient than navigating to the object for every change.

For example, suppose you want to optimize the Length input of the first strategy, and you want to test a list of discrete values on that input. The Optimization API supports this kind of optimization via the `OptList` property. If the list of input values is contained in a `Vector` of doubles called `values`, you could specify this optimization as follows:

```
method void OptimizeList(Vector values)
vars:
   tsopt.OptList optList,
   int j;
begin
   optList = job.Strategies[0].ELInputs.OptList("Length");
   for j = 0 to values.Count - 1 begin
      optList.AddValue(values[j] astype double);
   end;
end;
```

It would be inefficient to navigate from the `Job` object to the `OptList` object on each iteration of the loop. Saving the `OptList` object in a local variable solves this problem, and it also makes the code easier to read.

Optimizing Strategy Parameters

The Optimization API offers several different ways to optimize strategies:

1. You can optimize an input over a range of numbers. This is the method traditionally supported by TradeStation.
2. You can optimize an input over a list of values. This is useful for the following scenarios:
   a) You want to optimize over a discrete set of values that can't be represented by a range (e.g. 1, 3, 7, 15, 30).
   b) You want to optimize a text input by passing a list of different text values.
   c) You want to optimize an input that expects an expression (e.g. BollingerPrice). You can use a list to test multiple expressions for that input (e.g. "Close", "High", and "Low").
3. You can optimize a Boolean input by testing both True and False values.
4. You can optimize whether a strategy is enabled. This is useful when you are optimizing a set of cooperating strategies, and you want to turn some of them on or off in order to determine whether they make a useful contribution.

The following sections describe how to request these different kinds of optimization via the Optimization API.

Optimizing by Range

To optimize an input over a range of numbers, call the `ELInputs.OptRange` method, passing the name of the input, the start of the range, the end of the range, and the step value. For example, you could use the following code to optimize the Length input of the first strategy from 10 to 30 by 2:

```
job.Strategies[0].ELInputs.OptRange("Length", 10, 30, 2);
```

Optimizing by List

To optimize an input over a list of values, call the `ELInputs.OptList` method, passing the name of the input. The method returns an `OptList` object, which provides methods for adding, changing, or deleting values.

If you are using the traditional style, it's a good idea to save the `OptList` object in a variable to avoid repetition:

```
method void OptimizeBollingerPrice()
vars:
   tsopt.OptList optList;
begin
   optList = job.Strategies[0].ELInputs.OptList("BollingerPrice");
   optList.AddValue("Close");
   optList.AddValue("High");
   optList.AddValue("Low");
end;
```

If you are using the chained method style, you can chain `AddValue` methods together to populate the list:

```
job.Strategies[0].ELInputs.OptList("BollingerPrice")
   .AddValue("Close")
   .AddValue("High")
   .AddValue("Low");
```

Notice that if you are just chaining some method calls together and not following them with an `End...` property, you can omit the `UseDefinition()` call at the end.

Within a tree-style definition, you can use the `EndOptList` property to return to the context of the `ELInputs` object. This allows you to chain additional `ELInputs` methods:

```
job.Strategies[0]
   .ELInputs
      .OptList("BollingerPrice")
         .AddValue("Close")
         .AddValue("High")
         .AddValue("Low")
      .EndOptList
      .OptRange("Length", 10, 30, 2)
      .OptRange("NumDevsDn", 1, 3, 0.25)
   .EndELInputs
.UseDefinition();
```

To perform repeated updates on an `OptList`, save the object in a local variable so that you can access it efficiently:

```
method void OptimizeLength(Vector values)
vars:
   tsopt.OptList optList,
   int j;
begin
   optList = job.Strategies[0].ELInputs.OptList("Length");
   for j = 0 to values.Count - 1 begin
      optList.AddValue(values[j] astype double);
   end;
end;
```

Notice that you can add either numbers or expressions (or both) to an `OptList`. To add a number to the list, call the `AddValue` method and pass the number. To add an expression to the list, call the `AddValue` method and enclose the expression in quotes.

Note: It's fine to pass a string containing a number, since this is a valid expression. Thus, if you are getting the value from a `TextBox`, you don't need to convert it to a number before adding it to the `OptList` object. There is no difference between calling `AddValue("15")` and calling `AddValue(15)`.

You can also optimize over a list of text values. This is admittedly an unusual case, but if you wish to do so, you can use the `AddValueAsText` method in order to indicate that the string is a text value and not an expression. For example, suppose you have a strategy called "Method Test" that includes a "MethodName" text input. The strategy performs different operations based on the name of the method passed via the input. You could optimize over a list of method names as follows:

```
job.Strategies.AddStrategy("Method Test")
   .ELInputs
      .OptList("MethodName", tsopt.InputType.TextType)
         .AddValueAsText("ADX")
         .AddValueAsText("RSI")
         .AddValueAsText("Stochastics");
```

Note that you also need to specify that the input type is `TextType` in the optional second argument to `OptList`. The default input type is `NumericType`, so you can omit it for numeric inputs.

Optimizing Boolean Inputs

To optimize a Boolean input, call the `ELInputs.OptBool` method, passing the name of the input. You do not need to provide any additional information, since the optimizer will test only True and False values for the input:

```
job.Strategies[0].ELInputs.OptBool("UseProfitTarget");
```

Setting an Input to a Fixed Value

To set an input to a fixed value in the job definition, call the `ELInputs.SetInput` method, passing the name of the input, its value, and optionally its type (which defaults to Numeric). As with `OptList`, you can pass a numeric value, a numeric expression, or a text value for the input.

```
job.Strategies[0].ELInputs.SetInput("Length", 14);
```

To specify a text value, call the `SetInputAsText` method instead of `SetInput`:

```
job.Strategies[0].ELInputs.SetInputAsText("MethodName", "RSI");
```

If you do not optimize an input or set its value explicitly, the Optimization API will use the default value of the input.

Optimizing a Strategy's Enabled Status

If you have a group of cooperating strategies, you can optimize the Enabled status of one or more strategies to determine whether they make a useful contribution. Just call the `Strategy.OptStrategyEnabled` method for the strategy. For example, the following code optimizes the Enabled status of the third strategy:

```
job.Strategies[2].OptStrategyEnabled();
```

Notice that the method is called on the `Strategy` object, not the `ELInputs` sub-object, since it applies to the entire strategy.

Optimizing Security Parameters

The Optimization API also allows you to optimize securities. You can optimize the symbol and/or the interval for a given security (data series). You can even combine security optimization and strategy optimization within a single job.

Optimizing a Symbol

To optimize a symbol, use the `Security.OptSymbol` property instead of `SetSymbol`. The returned `OptSymbol` object provides an `AddSymbol` method for adding the symbols to test.

If you are using the traditional style, it's a good idea to save the `OptSymbol` object in a variable. You can then call `AddSymbol` multiple times to add symbols to the object. The optimizer will test each symbol in combination with any other optimized parameters. For example, the following code updates the first security to optimize over three different symbols:

```
method void OptimizeSymbols()
vars:
   tsopt.OptSymbol optSymbol;
begin
   optSymbol = job.Securities[0].OptSymbol;
   optSymbol.AddSymbol("AAPL")
   optSymbol.AddSymbol("CSCO")
   optSymbol.AddSymbol("MSFT");
end;
```

If you are using the tree style, you can just chain the `AddSymbol` calls:

```
job.Securities[0]
   .OptSymbol
      .AddSymbol("AAPL")
      .AddSymbol("CSCO")
      .AddSymbol("MSFT");
```

If the symbols are stored in an array or collection, you should save the `OptSymbol` object in a local variable for greater efficiency:

```
method void OptimizeSymbols(Vector symbols)
vars:
   tsopt.OptSymbol optSymbol,
   int j;
begin
   optSymbol = job.Securities[0].OptSymbol;
   for j = 0 to symbols.Count - 1 begin
      optSymbol.AddSymbol(symbols[j] astype string);
   end;
end;
```

In a tree-style job definition, you can use the `EndOptSymbol` property to return to the context of the `Security` object so that you can define other parts of the security:

```
job
   .Securities
      .AddSecurity()
         .OptSymbol
            .AddSymbol("AAPL")
            .AddSymbol("CSCO")
            .AddSymbol("MSFT")
         .EndOptSymbol
         .Interval
            .SetMinuteChart(5)
         .EndInterval
         .History
            .SetDaysBack(30)
         .EndHistory
      .EndSecurity
   .EndSecurities
UseDefinition();
```

Optimizing an Interval

To optimize an interval, use the `Security.OptInterval` property instead of `Interval`. The returned `OptInterval` object offers a variety of `Add…` methods for adding different types of intervals to test in the optimization. You can even mix intervals of completely different types, such as minute bars and Kagi bars. Just make sure that all of the intervals will work with a single history range. For example, you probably don't want to mix tick-based intervals and daily bars, since daily bars typically need a much longer history range than tick bars.

If you are using the traditional style, it's a good idea to save the `OptInterval` object in a variable. You can then call the relevant `Add…` methods multiple times to add intervals to the object. For example, the following code updates the first security to optimize over several different minute intervals:

```
method void OptimizeIntervals()
vars:
   tsopt.OptInterval optInterval;
begin
   optInterval = job.Securities[0].OptInterval;
   optInterval.AddMinuteChart(5)
   optInterval.AddMinuteChart(10)
   optInterval.AddMinuteChart(15);
end;
```

If you are using the tree style, you can chain the `Add…` methods together:

```
job.Securities[0]
   .OptInterval
      .AddMinuteChart(5)
      .AddMinuteChart(10)
      .AddMinuteChart(15);
```

The next example optimizes over several different advanced intervals:

```
job.Securities[0]
   .OptInterval
      .AddKagiChart(tsopt.Compression.Minute, 1,
                    tsopt.KagiReversalMode.FixedPrice, 0.1)
      .AddKaseChart(tsopt.Compression.Minute, 0.1)
      .AddRangeChart(tsopt.Compression.Minute, 1, 0.1);
```

If you are using a loop to add the intervals, you should save the `OptInterval` object in a local variable for greater efficiency. For example, the following code adds a series of minute intervals specified by the `startMinutes`, `stopMinutes`, and `step` arguments:

```
method void OptimizeIntervals(int startMinutes, int stopMinutes, int step)
vars:
   tsopt.OptInterval optInterval,
   int minutes;
begin
   optInterval = job.Securities[0].OptInterval;
   minutes = startMinutes;
   while minutes <= stopMinutes begin
      optInterval.AddMinuteChart(minutes);
      minutes += step;
   end;
end;
```

In tree-style job definitions, you can use the `EndOptInterval` property to return to the context of the `Security` object so that you can define other parts of the security.

Complex Optimizations over Symbols, Intervals, and Inputs

You can optimize symbols, intervals, and strategy inputs in a single job. The optimizer will test different combinations of all the optimized parameters. (In an Exhaustive optimization, the optimizer will test every possible combination of symbol, interval, and optimized input. In a Genetic optimization, the optimizer will test the combinations provided by the genetic algorithm.)

For example, the following complete job definition will optimize over two symbols, two intervals, and two strategy inputs:

```
method tsopt.Job DefineJob()
vars:
   tsopt.Job job,
   tsopt.Security security,
   tsopt.Strategy strategy;
begin
   job = new tsopt.Job;

   security = job.Securities.AddSecurity();

   security.OptSymbol.AddSymbol("AAPL");
   security.OptSymbol.AddSymbol("MSFT");

   security.OptInterval.AddMinuteChart(5);
   security.OptInterval.AddMinuteChart(10);
```

```
      security.History.LastDate = DateTime.Today;
      security.History.DaysBack = 30;

      strategy = job.Strategies.AddStrategy("Bollinger Bands LE");
      strategy.ELInputs.OptRange("Length", 10, 30, 1);

      strategy = job.Strategies.AddStrategy("Bollinger Bands SE");
      strategy.ELInputs.OptRange("Length", 10, 30, 1);

      return job;
   end;
```

Here's the same job definition in the tree style:

```
   job
      .Securities
         .AddSecurity()
            .OptSymbol
               .AddSymbol("AAPL")
               .AddSymbol("MSFT")
            .EndOptSymbol
            .OptInterval
               .AddMinuteChart(5)
               .AddMinuteChart(10)
            .EndOptInterval
            .History
               .SetLastDate(DateTime.Today)
               .SetDaysBack(30)
            .EndHistory
         .EndSecurity
      .EndSecurities
      .Strategies
         .AddStrategy("Bollinger Bands LE")
            .ELInputs
               .OptRange("Length", 10, 30, 1)
            .EndELInputs
         .EndStrategy
         .AddStrategy("Bollinger Bands SE")
            .ELInputs
               .OptRange("Length", 10, 30, 1)
            .EndELInputs
         .EndStrategy
      .EndStrategies
   .UseDefinition();
```

Specifying Job Settings

The Optimization API supports all of the available settings for TradeStation strategies. This includes settings that are specific to a particular strategy, as well as settings that apply to all of the strategies in an optimization.

Settings that apply to a particular strategy are defined within the relevant `Strategy` node. Settings that apply to all of the strategies (or to the optimization itself) are defined within the job's `Settings` node.

If you do not specify a setting in the job definition, the optimization will use the default value for that setting.

Since there are many global settings available, the `Settings` object groups them into a number of categories. Each category is represented by a different sub-options object, which can be accessed by a property with the same name. The Settings object provides the following sub-options properties:

- `GeneticOptions` (for genetic optimizations)
- `ResultOptions` (number of tests to keep, fitness function, etc.)
- `GeneralOptions` (base currency, MaxBarsBack, Look-Inside-Bar, etc.)
- `CostsAndCapital` (commission, slippage, capital, etc.)
- `PositionOptions` (pyramiding options and maximum shares per position)
- `TradeSize` (shares or currency per trade)
- `BackTesting` (fill options, market slippage, etc.)
- `OutSample` (size of the out-of-sample range, if any)
- `WalkForward` (enable/disable walk-forward optimization, name of walk-forward test)

Thus, to change a global setting, you must perform the following steps:

1. Get the `Settings` object from the `Job` object.
2. Get the relevant sub-options object from the `Settings` object.
3. Set the desired properties in the sub-options object.

You can often perform the first two steps in a single statement and save the sub-options object in a local variable. Then you can set the relevant properties on that object.

For example, suppose you have defined a job that performs a genetic optimization, and you want to change the genetic options for that job. Assuming that you have declared a local variable `geneticOptions` of type `tsopt.GeneticOptions`, you can do the following:

```
geneticOptions = job.Settings.GeneticOptions;
geneticOptions.Generations = 200;
geneticOptions.PopulationSize = 150;

// You can also read the current values of the options
crossoverRate = geneticOptions.CrossoverRate;
mutationRate = geneticOptions.MutationRate;
```

If you are setting just one property, you don't need to bother with saving the sub-options object in a local variable. Instead, you can perform the entire process in a single statement:

```
job.Settings.GeneticOptions.Generations = 200;
```

Of course, you can also omit the local variable when setting multiple sub-options, but it's less efficient and more wordy.

Each object that represents a group of options also offers a `Set…` method for each option. Thus, you can use chained methods to modify multiple options efficiently in a single statement:

```
job.Settings
   .GeneticOptions
      .SetGenerations(200)
      .SetPopulationSize(150);
```

Furthermore, each options object provides an `End…` property that returns its parent object in the job tree. This allows you to modify multiple groups of options in a single statement:

```
job.Settings
   .GeneticOptions
      .SetGenerations(200)
      .SetPopulationSize(150)
   .EndGeneticOptions
   .PositionOptions
      .SetPyramidingMode(tsopt.PyramidingMode.AnyEntry)
      .SetMaxSharesPerPosition(10000)
   .EndPositionOptions
UseDefinition();
```

See the `tsopt.Settings` object in the *API Reference* for a complete description of all the global settings.

See the `tsopt.Strategy` object in the *API Reference* for a description of strategy-specific settings.

Running an Optimization

Now that you know how to define an optimization job, you can assemble the other components of an optimization app.

The following sections explain each step of the process in detail.

Define the Optimization Job

First, you should write the code to define the optimization job, as described in the previous sections. You can hard-code the entire job definition if you want, but a UI-based application will probably assemble the job definition incrementally based on input supplied by the user.

It's also possible to load a job definition from an XML file. See the `Job` object in the *API Reference* for more information.

Define Event Handlers

When you run an optimization using the Optimization API, it executes asynchronously on its own threads. Thus, the API must provide a way to communicate with your application while the optimization is running and to notify your application when the optimization is done. The Optimization API provides this bridge via three events: `JobDone`, `JobFailed`, and `ProgressChanged`. You application must provide event handlers for the first two events, or you will get an error when you try to start the optimization. A handler for the `ProgressChanged` event is optional, but it is a good idea to provide one.

The following sections describe the purpose of each event and how to implement a handler for it.

The ProgressChanged Event

While an optimization is running, the Optimization API periodically sends a `ProgressChanged` event to provide information about the progress of the optimization. If you provide a handler for this event, you can show the progress of the optimization as it runs.

A `ProgressChanged` event handler has the following signature:

```
method void OptProgress(Object sender, tsopt.ProgressChangedEventArgs args)
begin
   // Code to handle event
end;
```

The `ProgressChangedEventArgs` class has two properties: `Progress` and `BestValues`.

The `Progress` property returns a `ProgressInfo` object, which provides information about the current state of the optimization, such as the test number, the elapsed time, and the remaining time. For a genetic optimization, it also provides information about the current generation and individual. See the *API Reference* for complete information about the `ProgressInfo` class.

The `BestValues` property returns a `BestValues` object, which provides information about the best result seen so far and the optimized parameters associated with that result. See the *API Reference* for complete information about the `BestValues` class.

Here is a simple implementation of a `ProgressChanged` handler. It assumes that the main form contains a `Label` called `statusLabel` that displays a status for the user. This method updates the status with the current test and the number of tests:

```
method void OptProgress(Object sender, tsopt.ProgressChangedEventArgs args)
begin
   statusLabel.Text = "Test " + args.Progress.TestNum.ToString()
      + " of " + args.Progress.TestCount.ToString();
end;
```

After defining the event handler, you must assign the handler to the event in your `tsopt.Optimizer` object:

```
optimizer.ProgressChanged += OptProgress;
```

The JobFailed Event

If an error occurs during an optimization, the Optimization API sends a `JobFailed` event. You *must* provide a handler for this event, since otherwise your application will have no way to know when an optimization halts with an error. (The optimization engine doesn't throw exceptions for errors while an optimization is running because it executes asynchronously in the background.) If you forget to provide a `JobFailed` event handler, the API will throw an `InvalidOperationException` when you start the optimization.

A `JobFailed` event handler has the following signature:

```
method void OptError(Object sender, tsopt.JobFailedEventArgs args)
begin
   // Code to handle event
end;
```

The `JobFailedEventArgs` class has a property called `Error`. This returns an instance of the `tsopt.OptimizationException` class. You can call the `Message` property on that object to get the error message.

Here is a simple implementation of a `JobFailed` handler. It assumes that the form contains a `Label` called `statusLabel`.

```
method void OptError(Object sender, tsopt.JobFailedEventArgs args)
begin
   statusLabel.Text = "Optimization Error: " + args.Error.Message;
end;
```

Note that you can easily report the error in the Events Log if you wish. Since the error information is provided within a `tsopt.OptimizationException` object, you can just throw the exception from your event handler:

```
method void OptError(Object sender, tsopt.JobFailedEventArgs args)
begin
   throw args.Error;
end;
```

After defining the event handler, you must assign the handler to the event in your `tsopt.Optimizer` object:

```
optimizer.JobFailed += OptError;
```

The *Error Handling* section below describes optimization errors in greater detail.

The JobDone Event

When an optimization stops normally (i.e., without an error), the Optimization API calls your `JobDone` event handler. There are two reasons why this handler can be called:

- The optimization completed successfully.
- The optimization was canceled by the user.

If your application provides a way to cancel an optimization, you may want to detect this case and display an appropriate status message. Otherwise, you can handle a completed optimization and a canceled optimization in the same way. Both cases allow you to retrieve and report the optimization results. If the optimization was canceled, the results will contain all the tests that were completed.

Most applications will keep their own record of whether the user canceled the optimization (probably in a top-level variable). You can check that variable to determine whether a cancellation occurred. You can also check the `Canceled` property of the `JobDoneEventArgs` object to determine whether the user canceled the optimization.

You *must* provide a `JobDone` event handler, since otherwise your application will have no way to know when the optimization is done. If you forget to provide a `JobDone` handler, the API will throw an `InvalidOperationException` when you start the optimization.

A `JobDone` event handler has the following signature:

```
method void OptDone(Object sender, tsopt.JobDoneEventArgs args)
begin
   // Code to handle event
end;
```

A typical `JobDone` event handler will perform the following tasks:

1. If the application provides cancellation as an option, update the status display to indicate whether the optimization completed or was canceled.
2. Get a `tsopt.Results` object from the `args.Result` property, and use it to display or save the optimization results. The `tsopt.Results` class is described in detail in a later section.

Here is a simple implementation of the `OptimizationDone` method. A more complete implementation would follow the same basic pattern as this example, but would typically provide more information about the optimization results.

```
method void OptDone(object sender, tsopt.JobDoneEventArgs args)
vars:
   tsopt.Results results;
begin
   results = args.Results;

   statusLabel.Text = "Optimization done";
   resultsText.Text =
      "Net Profit = " + NumToStr(results.NetProfit(), 2) + NewLine
      "Profit Factor = " + NumToStr(results.ProfitFactor(), 2) + NewLine +
      "% Profitable = " + NumToStr(results.PercentProfitable(), 2);
end;
```

After defining the event handler, you must assign the handler to the event in your `tsopt.Optimizer` object:

```
optimizer.JobDone += OptDone;
```

Start the Optimization

Once you have defined the job, created a `tsopt.Optimizer` object, and implemented your event handlers, the only remaining task is to start the optimization.

To run an optimization, call the `Optimizer.StartJob` method and pass it the job definition. You may also pass the number of threads to use as a second argument; if you omit it, the Optimization API will determine the thread count automatically based on the number of processor cores.

An optimization will typically be started in response to a button click or menu selection. Suppose your TradingApp contains a variable called `optimizer`, and suppose the job definition is stored in a

variable called `job`. If the form has a button named `StartButton`, we can write the following `Click` handler to start the optimization:

```
method void OnStartButtonClick(elsystem.Object sender, elsystem.EventArgs args)
begin
   optimizer.StartJob(job);
end;
```

If there are errors in the job definition, the API will throw a `tsopt.OptimizationException`, and the error will be reported in the Events Log. If you wish to handle the error yourself, you can wrap the `StartJob` call in a `try/catch` block and catch the `OptimizationException`. See the section on *Error Handling* below for more information.

Canceling an Optimization

If an optimization can run for a long time, it's a good idea to provide a way to cancel it. The Optimization API makes this easy to do: just call the `Optimizer.CancelJob` method while the optimization is running.

Note that the optimization may not stop immediately. The optimization engine will cancel the optimization as soon as it can, and then it will call your `JobDone` event handler to let you know that the optimization has stopped. Your handler can retrieve any results that were processed before the cancellation by accessing the `Results` property of the `args` argument.

Typically a TradingApp will provide a button to cancel the optimization, and the `Click` handler for that button will call the `CancelJob` method. If your application already has a button to start an optimization, a useful convention is to use the same button to cancel the optimization. If an optimization has not started yet, the button text can be "Start Optimization" (or whatever you prefer). When the optimization starts, you can change the button text to "Cancel Optimization". Finally, when the optimization finishes (either normally or because it was canceled), you can change the button text back to "Start Optimization" again.

The following code shows one way to implement this pattern:

```
vars:
   tsopt.Optimizer optimizer(null),
   tsopt.Job job(null),
   intrabarpersist bool started(false),
   intrabarpersist bool canceled(false);

method void OnStartButtonClick(elsystem.Object sender, elsystem.EventArgs args)
begin
   if not started then begin
      started = true;
      startButton.Text = "Cancel Optimization";
      StartOptimization();
   end
   else begin
      canceled = true;
      startButton.Enabled = false;
      optimizer.CancelJob();
   end;
end;
```

```
method void StartOptimization()
begin
   // Define job and start optimization here
end;

method void ResetOptimization()
begin
   canceled = false;
   started = false;

   startButton.Text = "Start Optimization";
   startButton.Enabled = true;
end;

method void OptError(Object sender, tsopt.JobFailedEventArgs args)
begin
   statusLabel.Text = "Optimization Error: " + args.Error.Message;
   ResetOptimization();
end;

method void OptDone(object sender, tsopt.JobDoneEventArgs args)
vars:
   tsopt.Results results;
begin
   results = args.Results;

   // Do something with the results

   if canceled then
      statusLabel.Text = "Optimization canceled"
   else
      statusLabel.Text = "Optimization done";

   ResetOptimization();
end;
```

Note that the sample above processes the results even if the optimization has been canceled. If you want to process results only if the optimization has completed, you can return early from the `OptDone` method when `canceled` is true:

```
method void OptDone(object sender, tsopt.JobDoneEventArgs args)
vars:
   tsopt.Results results;
begin
   if canceled then begin
      progressLabel.Text = "Optimization canceled";
      ResetOptimization();
      return;
   end;

   results = args.Results;

   // Do something with the results

   statusLabel.Text = "Optimization done";
   ResetOptimization();
end;
```

Error Handling

There are three categories of errors you may encounter when working with the Optimization API:

1. **Job Definition Errors.** These are non-recoverable errors within the code that defines a job. They are always coding errors, and since they prevent further definition of the job, the API reports them by throwing an exception.
2. **Job Validation Errors.** A job definition is validated for completeness and correctness when you start an optimization. The API reports validation errors by throwing an exception from the `StartJob` method. These are usually coding errors as well, but they can also be caused by problems in the environment (e.g., a strategy required by the optimization job is not defined in the current TradeStation environment).
3. **Runtime Errors**. These are errors that occur after starting the optimization. The API reports runtime errors by sending an event to your `JobFailed` handler.

The following sections describe how to handle each of these error types.

Catching Errors in Job Definitions

The code that defines an optimization job can throw exceptions for certain kinds of coding errors. The most common case is passing an invalid index into a collection. Since there is no way to proceed after this kind of error, the Optimization API will throw an `OptimizationException`.

Normally these errors will be displayed in the Events Log. However, if you are having trouble isolating an error, you may wish to wrap each section of job definition code in a `try/catch` block in order to log more information about the error. Since these are fatal errors, it's a good idea to re-throw the exception in order to halt execution of your application.

For example, the following code will trigger an exception because it attempts to index into the `Securities` object before a security has been added. The `catch` block outputs some information about the error to the Print Log; then it re-throws the exception in order to halt the application.

```
job = new tsopt.Job();
try
   job.Securities[0].SetSymbol("MSFT"); // this will throw an exception!
catch (tsopt.OptimizationException error)
   Print("Error setting symbol: ", error.Message);
   Print("   in API method: ", error.Source);
   throw error;
end;
```

Note that you should avoid reporting these errors directly to the user in a released application, since they are *always* coding errors. They are intended primarily for debugging purposes.

Handling Validation Errors

Most errors in a job definition are caught during the job validation phase, which is performed at the start of an optimization. The Optimization API reports these errors by throwing an exception from the `Optimizer.StartJob` method. Here are some common types of validation errors:

- The job definition is incomplete. For example, if you forget to define an interval for a security, this would be reported as a validation error.
- The job contains an invalid strategy name.
- The job contains an invalid input name.

The message for a validation error includes a "path" down the job definition tree to the node that generated the error. This makes it easier to locate the source of the error. For example, suppose a job definition contains an invalid input name. The generated error message might be something like this:

```
Strategies: Strategy[1: Bollinger Bands SE]: Inputs: Invalid input name: XYZ
```

Since the `StartJob` method throws an exception for a validation error, it will show up in the Events Log by default. If you wish to handle validation errors yourself, you can wrap the `StartJob` call in a `try/catch` block. For example, the following code displays a validation error in a `Label` control:

```
try
   optimizer.StartJob(job);
catch (tsopt.OptimizationError error)
   statusLabel.Text = "Invalid job: " + error.Message;
end;
```

Although the "error path" can be very helpful for debugging, you may wish to omit it when reporting validation errors to a user. (Most validation errors can be eliminated by correct coding, but they can also be caused by problems in the environment. For example, the current workspace may be missing a strategy that is used by an optimization.) To get the error message without the path, use the `error.MessageNoPath` property instead of `error.Message`.

Handling Runtime Errors

There are a number of errors that cannot be detected until an optimization is actually running. When a runtime error occurs, the Optimization API halts the optimization and sends the `JobFailed` event.

To handle a runtime error, perform the following steps in your `JobFailed` event handler:

1. Retrieve the error object from the `args.Error` property. Then use the error object's `Message` property to retrieve the error message.
2. Report the error to the user.

Note that runtime error messages never include a "path," so the `Message` and `MessageNoPath` properties will return the same value for those errors.

Summary of Error Handling

The Optimization API was designed to centralize error handling in order to make it as simple as possible for the client application. In summary, you just need to handle the following cases:

- Implement a `JobFailed` event handler to handle runtime errors. This handler should report the error message to the user.

- If you wish to handle validation errors yourself rather than letting them appear in the Events Log, wrap the call to `Optimizer.StartJob` in a `try/catch` block in order to catch the validation errors.
- If you need to debug certain kinds of coding errors in job definition code (e.g. invalid index errors), you can wrap this code in `try/catch` blocks as well.

Retrieving Optimization Results

When an optimization is done, an application can retrieve the results by accessing the `args.Results` property within its `JobDone` event handler. This returns a `tsopt.Results` object, which provides methods to get all of the information that is available in TradeStation's Strategy Optimization Report.

The examples above show how to fetch some simple metrics from the `Results` class, but this section will explore its capabilities in greater detail.

Strategy Metrics and tsopt.MetricID

An optimization in TradeStation calculates a number of different results that measure the performance of a strategy, such as Net Profit, Profit Factor, Percent Profitable, and so on. These are sometimes collectively called "fitness functions." More precisely, however, a fitness function is the *specific* measure of performance for which we are optimizing – i.e., if we optimize for Net Profit, *that* is the fitness function. Thus, the Optimization API uses the more general term "metric" to refer to any measure of strategy performance, whether or not it is used as the fitness function for a particular optimization.

The `tsopt.MetricID` enumeration provides an identifier for each of the available metrics. You can pass one of these identifiers to the `Results.GetMetric` method to get data for a particular metric. The `Results` class also provides named methods for all of the strategy metrics. The `GetMetric` method is more flexible, but in some cases the named methods can make your code more readable.

Structure of the Results

The optimization results are stored as a set of tests. Each test contains the following information:

- The test number. This identifies where the test appeared in the sequence of tests that were evaluated by the optimizer.
- The value of each optimized parameter for that test. An optimized parameter is usually an input, but it can also be a symbol, an interval, or a strategy's enabled status.
- The data for each strategy metric for that test. For each metric, the `Results` object stores a value for each combination of trade type (AllTrades, LongTrades, or ShortTrades) and range type (All, InSample, or OutSample).

In addition, the `Results` object provides some values that are not specific to any test, but that describe the optimization as a whole:

- The number of tests.
- The number of optimized parameters.

- The heading for each optimized parameter. The heading for an optimized *input* includes the strategy and input name, just like the Strategy Optimization Report. The heading for an optimized *symbol* is "DataN: Symbol" (where N is the number of the data series), and the heading for an optimized *interval* is "DataN: Interval".
- The heading for each strategy metric. Each heading includes the trade type (All, Long, or Short) and the metric name, just like the Strategy Optimization Report.

When you get data from a `Results` object, you may need to pass the parameter index and/or the test index of the value. If a method requires both of them, the parameter index always comes first. Note that these index values are zero-based.

When the `Results` object is retrieved from the `JobDoneEventArgs`, the tests are initially sorted by the fitness function that was used for the optimization. However, the `Results` class provides methods that make it easy to sort the tests by any metric as well as by test number.

Retrieving Information about Tests

To get the total number of tests in the `Results` object, use the `TestCount` property. This count will usually be the same as the number of results specified in the optimization settings (200 by default), but it may be less if the number of parameter combinations is lower than that setting. The number of tests can also be higher than the specified setting if there are multiple tests with identical fitness at the bottom of the results. (The `Results` object will include all of the "fitness ties" at the bottom, even if this causes the number of tests to be higher than requested. However, the number of tests will never exceed twice the requested number.) Thus, you should always use the `TestCount` property to determine the actual number of tests in the `Results` object.

To get the test number for a particular test, call the `GetTestNum` method and pass the index of the test.

Note: It is important to distinguish between the test *number* and the test *index*. The test *number* identifies where the test appeared in the sequence of all tests that were evaluated by the optimizer. The test *index* simply identifies which test to retrieve from the final set of results. For example, you can call `results.GetTestNum(2)` to get the test number for the third test. In this case, the test *index* is 2, whereas the method returns the test *number*, which is probably a completely different value. In fact, the test number can be much higher than `TestCount`, since it identifies the test among *all* the evaluated tests. The test index, however, should always be less than `TestCount`.

If you look at the Strategy Optimization Report in TradeStation, the test *number* is equivalent to the "Test" column in that report. The test *index* is equivalent to the row number, except that it starts at zero instead of one. Depending on how the results are sorted, the same test index can indicate different tests, whereas the test number for a particular test is always the same.

Retrieving Strategy Metrics

The Optimization API offers three basic techniques for getting strategy metrics from the `Results` object:

- Use a named metric method to get the specified metric for the first test. (If you have not re-sorted the tests, this will be the test with the best fitness value.)

- Use a named metric method to get the specified metric for a particular test index, trade type, and range.
- Use the `GetMetric` method to get a metric for a particular MetricID, test index, trade type, and range.

The following sections provide an example of each of these techniques.

Using a Named Method for the First Test

If you use a named metric method without any arguments, it will retrieve that metric for the first test in the results set, for all trade types and the entire history range. As mentioned above, the tests are initially sorted by fitness. Thus, if you have not re-sorted the tests, calling a named metric without arguments will give you the value for the "best" test.

For example, suppose you used Net Profit as the fitness function for the optimization. (This is the default.) Assuming that you have not re-sorted the tests, the following code will get the Profit Factor for the test that had the best Net Profit:

```
profitFactor = results.ProfitFactor();
```

Using a Named Method for a Specific Test, Trade Type, and Range

You can also call a named metric method and pass the test index, the trade type (AllTrades, LongTrades, or ShortTrades), and the results range (All, InSample, or OutSample). For example, the following code will get the Percent Profitable for the third test for long trades and in-sample results:

```
percentProfitable = results.PercentProfitable(2, tsopt.TradeType.LongTrades,
   tsopt.ResultsRange.InSample);
```

Using the GetMetric Method

You can call the `GetMetric` method to get any available metric for any test, trade type, and results range. The first argument is a `tsopt.MetricID` value that specifies the metric you wish to retrieve. The remaining arguments are the test index, trade type, and results range.

For example, the following code will get the Gross Profit for the fifth test for short trades and out-of-sample results:

```
grossProfit = results.GetMetric(tsopt.MetricID.GrossProfit, 4,
   tsopt.TradeType.ShortTrades, tsopt.ResultsRange.OutSample);
```

The `GetMetric` method is more verbose than the named metric methods, but it is also more flexible. Since the desired metric is passed as an argument, you can loop through a set of MetricIDs and pass them to the method to get the value for each metric. The extended example later in this section includes an example of this technique.

Retrieving Optimized Parameter Data

The strategy metrics are one key part of the results for a test; the other key part is the values of the optimized parameters that generated those metrics.

For example, suppose an application optimizes the FastLen and SlowLen inputs to a "MovAvg Cross" strategy. In order for the Net Profit in a particular test to be meaningful, you need to know the values of FastLen and SlowLen that were used for that test.

The following methods in the `Results` class provide information about the optimized parameters:

- The `OptParamCount` property returns the number of optimized parameters.
- The `GetOptParamHeading` method returns the descriptive heading for a particular optimized parameter. Since this is not specific to any particular test, the method accepts only the parameter index. In our moving average example, the first parameter heading would be "MovAvg Cross: FastLen" and the second parameter heading would be "MovAvg Cross: SlowLen".
- The `GetOptValueString` method returns a string representation of the value for a particular optimized parameter and test. This method expects two arguments: the parameter index and the test index.A string is returned because it is flexible enough to represent any optimized parameter, including symbols, intervals, and expression inputs. In our moving average example, the method will return values like "7" or "15". For an optimized symbol, it might return values like "CSCO" or "MSFT". For an optimized interval, it might return values like "5 min" or "10 min".

The extended example later in this section demonstrates how to use these methods.

Retrieving Specific Kinds of Parameter Data

The `GetOptParamHeading` and `GetOptParamString` methods are useful when you want to retrieve general information about all the optimized parameters for a job. However, there may be times when you want to get a specific kind of parameter for a test; for example, you may want to know the symbol that was used for a particular test, or you may want to know the value of a particular input. Although you could iterate through all the parameter headings and parse the strings to determine what type of data they represent (symbol, interval, input, etc.), there is a much easier way. The `tsopt.Results` class provides the following methods to retrieve a specific kind of parameter data for a particular test:

- The `GetTestSymbol` method returns the symbol that was used for a test. The first argument is the test index. For multi-data strategies, you may also pass an optional second argument that indicates the data series for the symbol (1 for Data1, 2 for Data2, etc.). If you omit the second argument, the method returns the symbol for Data1.
- The `GetTestInterval` method returns the interval that was used for a test. As with `GetTestSymbol`, the first argument is the test index, and the optional second argument indicates the data series. Note that this method returns a `tsopt.Interval` object, not a string. If you want to get a string representation of the interval, just call the `Describe` method on the returned `Interval` object. You may also call other methods or properties on the `Interval` object to get detailed information about the interval.
- The `GetTestInputString` method returns the string representation of a specific input value that was used for a test. You must pass the test index, the input name, and the strategy name as the first three arguments. If your job definition contains multiple instances of the same strategy, you should also pass a fourth argument that indicates which instance to use (0 for the first instance of that strategy, 1 for the second instance, and so on). Since this method returns the value as a string, it is

flexible enough to handle any kind of input, including text inputs, expression inputs, and Boolean inputs.

- The `GetTestInputDouble` method works just like `GetTestInputString`, but it returns the input value as a double, saving you the trouble of converting a string to a number. You should use this method only if you know that the specified input is numeric; otherwise, the method will throw an exception to indicate that it cannot convert the input value to a double.
- The `IsTestStrategyEnabled` method indicates whether a particular strategy was enabled for a test. This can be helpful if you used the `OptStrategyEnabled` method to optimize the enabled status of a strategy, and you want to know whether the strategy was enabled or disabled for a particular test. You must pass the test index and strategy name as the first two arguments. If your job definition contains multiple instances of the same strategy, you should also pass a third argument that indicates which instance to use (0 for the first instance, 1 for the second instance, and so on).

Here are some examples of these methods:

```
sym = results.GetTestSymbol(0); // get the symbol used for the first test

sym = results.GetTestSymbol(4, 2); // get symbol for the fifth test for Data2

interval = results.GetTestInterval(3); // get the interval for the fourth test

// Get value of "Length" input for "Bollinger Bands LE" strategy for first test
nInput = results.GetTestInputDouble(0, "Length", "Bollinger Bands LE");

// Get value of "Bollinger Price" input for "Bollinger Bands LE" for first test
sInput = results.GetTestInputString(0, "Bollinger Price", "Bollinger Bands LE");
```

One important feature of these methods is that they work for both optimized and non-optimized parameters. For example, the `GetTestInputDouble` example above will retrieve the value of the "Length" input whether or not that input is optimized. If the input is optimized, the returned value may vary between different tests. If the input is not optimized, the returned value for that input will be the same for all tests.

Sorting the Results

The `Results` class provides two methods to sort the tests:

- The `SortByTestNum` method sorts the tests by test number.
- The `SortByMetric` method sorts the tests by a specified metric, trade type, and results range.

Both methods allow you to sort in either ascending or descending order. To sort in descending order, pass true for the `reverse` argument.

For example, the following code sorts the results by test number in ascending order:

```
results.SortByTestNum(false);
```

The following code sorts the results by Profit Factor in descending order (from largest to smallest values). It uses the in-sample results for all trades (long and short):

```
results.SortByMetric(tsopt.MetricID.ProfitFactor, tsopt.TradeType.AllTrades,
   tsopt.ResultsRange.InSample, true {reverse});
```

Example: Writing the Results to a Text File

The following example demonstrates many of the key features of the `Results` class. It writes the optimization results for a specified trade type and results range to a comma-delimited text file. The format of the file is very similar to the Strategy Optimization Report in TradeStation.

Important Note: The `tsopt.Results` class already provides a `WriteFile` method that can produce the same file as the code below. However, you may still find the code below useful, both as an example of how to use the `tsopt.Results` class, and as a template in case you want to save your own customized version of the optimization results. The `Results.WriteFile` method is discussed after the example.

This method could be called from the `JobDone` event handler to save the results of an optimization.

```
method void WriteResults(string path, tsopt.Results results,
                         tsopt.TradeType type, tsopt.ResultsRange range)
vars:
   elsystem.io.StreamWriter file,
   int paramCount,
   int param,
   int metricID,
   int index,
   double metric,
   string s,
   string delim;
begin
   file = elsystem.io.StreamWriter.Create(path);

   delim = ",";

   // Write the column headings for the report
   // First, write the optimized parameter headings
   paramCount = results.OptParamCount;
   for param = 0 to paramCount - 1 begin
      if param > 0 then      // If past the first column
         file.Write(delim);  //    write a delimiter
      file.Write(results.GetOptParamHeading(param));
   end;

   // Next, write the heading for the test number
   file.Write(delim);
   file.Write("Test");

   // Finally, write a heading for each strategy metric
   for metricID = 0 to tsopt.MetricID.MetricCount - 1 begin
      file.Write(delim);
      file.Write(tsopt.Results.GetMetricHeading(metricID, type));
   end;
   file.WriteLine(); // End the headings line
```

```
      // Now iterate through the tests and write the results
      for index = 0 to results.TestCount - 1 begin
         // First, write the optimized parameter values for this test
         for param = 0 to paramCount - 1 begin
            if param > 0 then     // If past the first column
               file.Write(delim); //    write a delimiter
            file.Write(CSV_Field(results.GetOptValueString(param, index)));
         end;

         // Next, write the test number for this test
         file.Write(delim);
         file.Write(results.GetTestNum(index));

         // Finally, write the value for each strategy metric
         for metricID = 0 to tsopt.MetricID.MetricCount - 1 begin
            metric = results.GetMetric(metricID, index, type, range);
            s = NumToStr(metric, 7); // format with 7 decimals
            file.Write(delim);
            file.Write(s);
         end;

         file.WriteLine(); // End this line of test results
      end;
   end;

   method string CSV_Field(string s)
   begin
      if InStr(s, ",") = 0 and InStr(s, DoubleQuote) = 0 then
         return s
      else
         return DoubleQuote + EscapeQuotes(s) + DoubleQuote;
   end;

   method string EscapeQuotes(string s)
   vars:
      string outStr,
      int j;
   begin
      outStr = "";
      j = InStr(s, DoubleQuote);
      while j <> 0 begin
         outStr += LeftStr(s, j) + DoubleQuote;
         s = MidStr(s, j + 1, 999999);
         j = InStr(s, DoubleQuote);
      end;
      return outStr + s;
   end;
```

Notice that the example uses the `GetMetric` method to retrieve the metrics data. Since this method accepts a `tsopt.MetricID` to identify the desired metric, the code can iterate through all the metrics and write out their results. This is much more concise than calling a named method (e.g., `NetProfit`, `GrossProfit`, etc.) for each metric.

The code also calls the static `Results.GetMetricHeading` method to get a heading for each `MetricID`. This method prefixes each metric name with the trade type, just like the headings in the Strategy Optimization Report. To get the metric name without the trade type prefix, call the `Results.GetMetricName` method instead.

It's possible for the `GetOptValueString` method to return a string that contains commas or quotes (e.g., if we are optimizing over expressions or text values). Thus, we also provide a helper method that returns a valid CSV field for these cases. This method ensures that a field that contains a comma or quote will be surrounded by quotes, and that any embedded quotes will be escaped (by converting them into two quote marks). We pass the return value of `GetOptValueString` to `CSV_Field` and write the converted string to the file. This allows the file to be properly parsed and opened by tools such as Microsoft Excel®.

Writing Results the Easy Way

If you want to write your results to a file that has the same basic format as the Strategy Optimization Report, you don't have to write your own code to do it. The `tsopt.Results` class includes a `WriteFile` method that will do all the work for you.

The `Results.WriteFile` method has the following signature:

```
void WriteFile(string fileName, string delimiter,
   tsopt.TradeType type, tsopt.ResultsRange range);
```

If you pass a comma as the delimiter, this method will produce a valid CSV file.

Here is a simple `JobDone` event handler that uses this method to write out the results:

```
method void OptDone(object sender, tsopt.JobDoneEventArgs args)
vars:
   tsopt.Results results;
begin
   results = args.Results;

   results.WriteFile("C:\Logs\OptResults.csv", ",",
      tsopt.TradeType.AllTrades, tsopt.ResultsRange.All);

   Print("Optimization done");
   Print("Net Profit = ", results.NetProfit());
end;
```

Queued Optimizations

Many applications will need to run only a single optimization at a time. However, there may be occasions when you want to set up multiple optimizations and then run them all in sequence. The Optimization API makes this very easy to do.

Suppose that you have defined two optimization jobs, which are stored in variables called `job1` and `job2`. You can automatically run the two jobs in sequence by calling the `StartJob` method for each job:

```
optimizer.StartJob(job1);
optimizer.StartJob(job2);
```

This kicks off the following sequence of events:

1. The optimizer immediately starts optimizing `job1` (using all available cores), and it places `job2` into its job queue.
2. When `job1` finishes optimizing, the optimizer calls your `JobDone` event handler so that you can process the results. Then it starts optimizing `job2`.
3. When `job2` finishes optimizing, the optimizer calls your `JobDone` event handler again so that you can process the results for the second job.

The Job ID

Since the optimizer calls your `JobDone` event handler twice in the example above, you may be wondering how you can tell which job has just finished.

The answer is provided by the `JobID` property of the `args` argument in your event handler. This is a 64-bit integer that uniquely identifies each queued job within an application. You can access this property to determine which job has just finished.

The JobID is assigned when you start an optimization job, and it is returned by the `StartJob` method. If you are not queuing multiple jobs, you can safely ignore the return value, as we have done in previous examples. However, when you are using the queuing functionality, you may want to assign the value returned by `StartJob` to a variable. For example, the following code stores the IDs in variables called IDJob1 and IDJob2. The event handler then compares the `JobID` property to each variable to determine which job has completed.

```
vars:
   int64 IDJob1(0),
   int64 IDJob2(0);

method void StartOptimization()
begin
   IDJob1 = optimizer.StartJob(job1);
   IDJob2 = optimizer.StartJob(job2);
end;

// JobDone event handler
method void OptDone(Object sender, JobDoneEventArgs args)
begin
   if args.JobID = IDJob1 then begin
      // process results for job1
   end
   else if args.JobID = IDJob2 then begin
      // process results for job2
   end;
end;
```

A more general method for storing JobIDs might use a `Dictionary` object to associate each ID with a name. The following example uses two dictionaries: one to associate each name with a job definition, and another to associate each JobID with a name. This makes it easy to get the name for a JobID, and then to get the job definition for that name (if required).

```
vars:
   Dictionary dictJobDefs(null),
   Dictionary dictJobNames(null);

method void StartOptimization()
vars:
   Vector keys,
   tsopt.Job job,
   string jobName,
   int64 jobID,
   int j;
begin
   dictJobDefs = new Dictionary;  // associates names with job definitions
   dictJobNames = new Dictionary; // associates JobIDs with names

   job = new tsopt.Job;
   // Define Bollinger Bands optimization job here
   dictJobDefs["Bollinger Bands"] = job;

   job = new tsopt.Job;
   // Define Keltner Channel optimization job here
   dictJobDefs["Keltner Channel"] = job;

   // Iterate through the keys in dictJobDefs and start each job
   keys = dictJobDefs.Keys;
   for j = 0 to keys.Count - 1 begin
      jobName = keys[j] astype string;
      job = dictJobDefs[jobName] astype tsopt.Job;

      jobID = optimizer.StartJob(job);

      // Associate the job's name with its JobID
      dictJobNames[jobID.ToString()] = jobName;
   end;
end;
```

The `JobDone` event handler can now use these dictionaries to retrieve meaningful information about the completed job. For example, the following event handler retrieves the job name and uses it to construct a file name; then it writes the optimization results to that file.

```
// JobDone event handler
method void OptDone(Object sender, JobDoneEventArgs args)
vars:
   string jobName,
   string fileName;
begin
   jobName = dictJobNames[args.JobID.ToString()] astype string;
   fileName = "C:\Logs\Results - " + jobName + ".csv";

   args.Results.WriteFile(fileName, ",", tsopt.TradeType.AllTrades,
      tsopt.ResultsRange.All);
end;
```

As this example shows, it is not always necessary to write different code for different JobIDs. If you are handling the results for all the optimizations in the same way, you can just write the logic once in your `JobDone` event handler and let it handle all the jobs.

Although we have focused on the `JobDone` event handler, the `args` objects for the `JobFailed` and `ProgressChanged` event handlers also have a `JobID` property. You can use this property to determine which job triggered the event.

Canceling Queued Optimizations

When you are queuing multiple optimizations, there are three ways to cancel jobs:

- Call the `CancelJob` method with no arguments. The optimizer will cancel the currently running job and call your `JobDone` event handler for that job. If there are any jobs left in the queue, the optimizer will start the next job.
- Call the `CancelJob` method and pass the JobID of the job you want to cancel. If the specified job is currently running, the optimizer will cancel it, call your `JobDone` event handler, and start the next job (if any). Otherwise, the optimizer will remove the specified job from the queue.
- Call the `CancelAllJobs` method. The optimizer will cancel the currently running job and call your `JobDone` event handler for that job. If there are any jobs left in the queue, the optimizer will remove them.

API Reference

This reference documents all the classes in the Optimization API. It is divided into three sections:

- Job Definition Classes
- Optimization Classes
- The Results Class

Job Definition Classes

DefinitionObject Class

The `DefinitionObject` class is the parent of all the job definition classes. It provides a single method that is inherited by all of these classes:

`void UseDefinition();`

This method allows you to convert any tree-style job definition into a valid EasyLanguage statement. A job definition cannot end with one of the `End...` properties, since EasyLanguage does not allow you to end a statement with a property (unless it is being assigned to a variable). However, you can just add a `UseDefinition()` call after the last `End...` property, and this will make the definition a valid statement.

If a tree-style job definition ends with a method call rather than an `End...` property, the `UseDefinition()` call is optional.

Job Class

The `Job` class contains the definition of an optimization job. It provides the following members.

`static tsopt.Job Create();`

The default `Job` constructor creates a new job definition. It contains Securities, Strategies, and Settings nodes, but they are all empty. In order to run an optimization, you must define at least one valid security and one valid strategy.

`static tsopt.Job Create(string filename);`

This `Job` constructor loads the job definition from an XML file. The file would normally be created by calling `WriteXML` on a `Job` object.

`Object Clone();`

Creates a copy of the job definition. Since `Clone` is a standard method of the `Object` class, it returns the job as an `Object`. In order to use the cloned job, you will need to cast it to `tsopt.Job` as follows:

```
clonedJob = job.Clone() astype tsopt.Job;
```

property tsopt.OptimizationMethod OptimizationMethod { read; write; }

Gets or sets the optimization method, which can have one of the following values:

- `tsopt.OptimizationMethod.Exhaustive`
- `tsopt.OptimizationMethod.Genetic`

tsopt.Job SetOptimizationMethod(tsopt.OptimizationMethod optMethod);

Sets the optimization method for a job. Unlike the `OptimizationMethod` property, a `SetOptimizationMethod` call can be used in a tree-style job definition, since it returns the `Job` object.

property tsopt.Securities Securities { read; }

Returns a `Securities` object which represents the securities to use in the optimization job. Each security is equivalent to one of the data series in a TradeStation chart (Data1, Data2, etc.).

property tsopt.Strategies Strategies { read; }

Returns a `Strategies` object which represents the strategies to use in the optimization job.

property tsopt.Settings Settings { read; }

Returns a `Settings` object which represents the settings to apply to the entire optimization. Note that settings for a specific strategy are accessed using a `Strategy` object rather than the `Settings` object.

void WriteXML(string fileName);

Writes the job definition to an XML file with the specified file name. This definition can then be loaded at a later time by constructing a `Job` object with the file name.

Securities Class

The `Securities` class defines all the securities to use for an optimization. Each security is equivalent to a data series in a TradeStation chart (Data1, Data2, etc.). Many optimizations will use just one security, but a job can define multiple securities if it needs to optimize a multi-data strategy.

property tsopt.Job EndSecurities { read; }

Returns the parent object in the job definition tree, which is a `Job` object. This property is provided in order to support tree-style definitions.

property int Count { read; }

Returns the number of securities in the definition.

property default tsopt.Security Security[int] { read; }

Gets the `Security` object at the specified index. The `Securities` object represents a collection of `Security` objects, and you can access a specific security with array syntax. For example, the following code gets the first security in the job definition:

```
security = job.Securities[0];
```

tsopt.Security AddSecurity();

Adds a security to the job definition and returns the corresponding `Security` object. You can then call methods on that object to define the properties of the security.

tsopt.Securities DeleteSecurity(int pos);

Deletes the Security object at the specified index from the job definition.

bool ValidateSymbols(bool removeInvalidSymbols);

Checks all the securities in the job definition and verifies that their symbols are valid.

If the `removeInvalidSymbols` argument is false, the method will not modify the job definition, and it will return false if any symbols are invalid.

If the `removeInvalidSymbols` argument is true, the method will remove any securities with invalid symbols from the job definition. If there are any valid securities remaining, the method will return true; otherwise, it will return false.

bool ValidateSymbols(bool removeInvalidSymbols,
elsystem.collections.Vector invalidSymbols);

Checks all the securities in the job definition and verifies that their symbols are valid.

The behavior is the same as the first version of `ValidateSymbols`, but this version also adds any invalid symbols to the `invalidSymbols` vector. You must create the `Vector` object and pass it to the method.

Security Class

The `Security` class contains the definition of a single security within an optimization job. The primary components of a security are the symbol, interval, and history range. The symbol and the interval can be optimized.

property tsopt.Securities EndSecurity { read; }

Returns the parent object in the job definition tree, which is a `Securities` object. This property is provided in order to support tree-style definitions.

property bool IsSymbolOptimized { read; }

Indicates whether the symbol in the security is optimized (i.e., whether the symbol was specified via `OptSymbol` rather than via `Symbol` or `SetSymbol`).

property bool IsIntervalOptimized { read; }

Indicates whether the interval in the security is optimized (i.e., whether the interval was specified via `OptInterval` rather than via `Interval`).

property string Symbol { read; write; }

Gets or sets the symbol for the security.

If the symbol is currently optimized for a security, setting the `Symbol` property will change the job definition to use a fixed symbol.

tsopt.Security SetSymbol(string symbol);

Sets the symbol for the security. Unlike the `Symbol` property, a `SetSymbol` call can be used in a tree-style job definition, since it returns the `Security` object.

If the symbol is currently optimized for a security, calling the `SetSymbol` method will change the job definition to use a fixed symbol.

property tsopt.Interval Interval { read; }

Gets an `Interval` object that represents the interval for the security. You can call a method on the `Interval` object to set the desired interval.

If the interval is currently optimized for a security, modifying the returned `Interval` object will change the job definition to use a fixed interval.

If you want to convert an optimized interval to a fixed interval by choosing the first interval definition from the list of optimized intervals, call `ConvertOptIntervalToInterval` instead.

property tsopt.History History { read; }

Gets a `History` object that represents the history range for the security. You can call methods on the `History` object to set the desired range.

property tsopt.OptSymbol OptSymbol { read; }

Gets an `OptSymbol` object that represents an optimized symbol for the security. You can call methods on the `OptSymbol` object to define the list of symbols to optimize. See the *OptSymbol Class* entry below for more information.

If the security currently uses a fixed symbol, modifying the returned `OptSymbol` object will replace the fixed symbol with an optimized symbol. If you want to use the fixed symbol as the first item in a list of optimized symbols, call `ConvertSymbolToOptSymbol` instead.

property tsopt.OptInterval OptInterval { read; }

Gets an `OptInterval` object that represents an optimized interval for the security. You can call methods on the `OptInterval` object to define the list of intervals to optimize. See the *OptInterval Class* entry below for more information.

If the security currently uses a fixed interval, modifying the returned `OptInterval` object will replace the fixed interval with an optimized interval. If you want to use the fixed interval as the first item in a list of optimized intervals, call `ConvertIntervalToOptInterval` instead.

property tsopt.SecurityOptions SecurityOptions { read; }

Gets the `SecurityOptions` object for the security. This allows you to get or set the session name, volume usage, bar building, and time zone options for the security.

See the *SecurityOptions Class* entry below for more information.

tsopt.OptSymbol ConvertSymbolToOptSymbol();

Converts a fixed symbol to an optimized symbol by using the fixed symbol as the first item in the list of optimized symbols. The method returns the `OptSymbol` object so that additional symbols can be added to the list.

If no symbol is defined yet for the security, this method will still return an `OptSymbol` object, but the list of symbols will be empty.

string ConvertOptSymbolToSymbol();

Converts an optimized symbol to a fixed symbol by using the first optimized symbol as the fixed symbol. If the list of optimized symbols is empty, this method will throw an exception.

tsopt.OptInterval ConvertIntervalToOptInterval();

Converts a fixed interval to an optimized interval by using the first interval as the first item in the list of optimized intervals. The method returns the `OptInterval` object so that additional intervals can be added to the list.

If no interval is defined yet for the security, this method will still return an `OptInterval` object, but the list of intervals will be empty.

tsopt.Interval ConvertOptIntervalToInterval();

Converts an optimized interval to a fixed interval by using the first optimized interval as the fixed interval. If the list of optimized intervals is empty, this method will throw an exception.

void EnsureHistoryCompatibleWithInterval();

Ensures that the current history range for the security is compatible with the interval definition. For example, if the interval is defined as Weekly, then the history range cannot be based on days. If the history definition is incompatible, this method will change it to a compatible definition, using the default range for the current interval.

If the security has an optimized interval, this method will ensure that the history definition is compatible with all of the intervals in the list.

`string DescribeSymbol();`

Returns a string that describes the symbol definition. For a fixed symbol, this method will simply return the symbol. For an optimized symbol, this method will return a comma-delimited list of symbols (e.g. “MSFT, AAPL”).

`string DescribeInterval();`

Returns a string that describes the interval definition. For example, a 5 minute interval would be described as “5 min”. For an optimized interval, this method will return a list of intervals (e.g. “5 min; 10 min”). The list is delimited by semicolons because some interval descriptions can contain commas.

`string DescribeHistory();`

Returns a string that describes the history range for the security (e.g. “Last Date: 12/31/2012, 30 days back”).

Interval Class

The `Interval` class defines the interval for a security. It provides methods to set different kinds of intervals, as well as properties that return information about the current interval definition.

`property tsopt.Security EndInterval { read; }`

Returns the parent object in the job definition tree, which is a `Security` object. This property is provided in order to support tree-style definitions.

`tsopt.Interval SetTickChart(int ticks);`

Sets a tick-based interval with the specified number of ticks per bar.

`tsopt.Interval SetVolumeChart(int shares);`

Sets a volume-based interval with the specified number of shares per bar.

`tsopt.Interval SetSecondChart(int seconds);`

Sets a second-based interval with the specified number of seconds per bar.

`tsopt.Interval SetMinuteChart(int minutes);`

Sets a minute-based interval with the specified number of minutes per bar.

`tsopt.Interval SetDailyChart();`

Sets a daily interval.

`tsopt.Interval SetWeeklyChart();`

Sets a weekly interval.

`tsopt.Interval SetMonthlyChart();`

Sets a monthly interval.

`tsopt.Interval SetKagiChart(tsopt.Compression resolutionUnit, int resolutionQty, tsopt.KagiReversalMode reversalMode, double reversalSize);`

Sets a Kagi interval based on the underlying time interval specified by `resolutionUnit` and `resolutionQty`. The `reversalMode` and `reversalSize` arguments determine how the bars are built.

The `reversalMode` argument can be either `tsopt.KagiReversalMode.FixedPrice` or `tsopt.KagiReversalMode.Percent`.

`tsopt.Interval SetKaseChart(tsopt.Compression resolutionUnit, double targetRange);`

Sets a Kase interval based on the underlying time interval specified by `resolutionUnit`. The resolution quantity for Kase bars is always 1. The `targetRange` argument determines how the bars are built.

`tsopt.Interval SetLineBreakChart(tsopt.Compression resolutionUnit, int resolutionQty, int lineBreaks);`

Sets a Line Break interval based on the underlying time interval specified by `resolutionUnit` and `resolutionQty`. The `lineBreaks` argument determines how the bars are built.

`tsopt.Interval SetMomentumChart(tsopt.Compression resolutionUnit, int resolutionQty, double range);`

Sets a Momentum interval based on the underlying time interval specified by `resolutionUnit` and `resolutionQty`. The `range` argument determines how the bars are built.

`tsopt.Interval SetPointFigureChart(tsopt.Compression resolutionUnit, int resolutionQty, double boxSize, int reversal, tsopt.PointFigureBasis basis, bool enableOneStepBack);`

Sets a Point-and-Figure interval based on the underlying time interval specified by `resolutionUnit` and `resolutionQty`. The other arguments determine how the bars are built.

The `basis` argument can be either `tsopt.PointFigureBasis.Close` or `tsopt.PointFigureBasis.HighLow`.

`tsopt.Interval SetRangeChart(tsopt.Compression resolutionUnit, int resolutionQty, double range);`

Sets a Range Chart interval based on the underlying time interval specified by `resolutionUnit` and `resolutionQty`. The `range` argument determines how the bars are built.

`tsopt.Interval SetRenkoChart(double brickSize);`

Sets a Classic Renko interval, which always uses an underlying interval of 1 tick. The `brickSize` argument determines how the bars are built.

`tsopt.Interval SetMeanRenkoChart(double brickSize);`

Sets a Mean Renko interval, which always uses an underlying interval of 1 tick. The `brickSize` argument determines how the bars are built.

`tsopt.Interval SetSpectrumRenkoChart(double brickSize, double brickOffset);`

Sets a Spectrum Renko interval, which is equivalent to a Custom Renko interval in Charting. This interval always uses an underlying interval of 1 tick. The `brickSize` and `brickOffset` arguments determine how the bars are built.

`property tsopt.ChartType ChartType { read; }`

Returns the chart type of the current interval. This will be one of the following values:

- `tsopt.ChartType.Bar` (for time-based charts and tick charts)
- `tsopt.ChartType.Volume`
- `tsopt.ChartType.Kagi`
- `tsopt.ChartType.Kase`
- `tsopt.ChartType.LineBreak`
- `tsopt.ChartType.Momentum`
- `tsopt.ChartType.PointAndFigure`
- `tsopt.ChartType.Range`
- `tsopt.ChartType.Renko`

`property tsopt.Compression ResolutionUnit { read; }`

For time-based charts and tick charts, this returns the basic unit of the interval (Tick, Second, Minute, Daily, Weekly, or Monthly).

For advanced intervals, this returns the basic unit of the underlying time or tick interval that is used to build the bars.

`property int ResolutionQty { read; }`

For time-based charts and tick charts, this returns the number of units per interval. For example, a 5 minute chart would have a `ResolutionUnit` of `tsopt.Compression.Minute` and a `ResolutionQty` of 5.

For advanced intervals, this returns the number of units in the underlying time or tick interval that is used to build the bars. For example, if a Kagi interval is based on 10 tick bars, then `ResolutionUnit` would return `tsopt.Compression.Tick` and `ResolutionQty` would return 10.

`property bool IsAdvancedInterval { read; }`

Indicates whether the current interval is an advanced interval (Kagi, Kase, LineBreak, Momentum, PointAndFigure, Range, or Renko).

`property double PriceRange { read; }`

Returns the price range used to build Kase, Momentum, or Range bars. If the interval is not one of these chart types, this property returns zero.

`property tsopt.KagiReversalMode KagiReversalMode { read; }`

Returns the reversal mode for a Kagi interval. This property should only be called if the `ChartType` is `tsopt.ChartType.Kagi`; otherwise, its value is not meaningful.

The value returned by this property can be either `tsopt.KagiReversalMode.FixedPrice` or `tsopt.KagiReversalMode.Percent`.

`property double KagiReversalSize { read; }`

Returns the reversal size for a Kagi interval. This property should only be called if the `ChartType` is `tsopt.ChartType.Kagi`; otherwise, its value is not meaningful.

`property int LineBreaks { read; }`

Returns the number of line breaks for a Line Break interval. This property should only be called if the `ChartType` is `tsopt.ChartType.LineBreak`; otherwise, its value is not meaningful.

`property double PointFigureBoxSize { read; }`

Returns the box size for a Point-and-Figure interval. This property should only be called if the `ChartType` is `tsopt.ChartType.PointAndFigure`; otherwise, its value is not meaningful.

`property int PointFigureReversal { read; }`

Returns the reversal value for a Point-and-Figure interval. This property should only be called if the `ChartType` is `tsopt.ChartType.PointAndFigure`; otherwise, its value is not meaningful.

`property tsopt.PointFigureBasis PointFigureBasis { read; }`

Returns the basis for a Point-and-Figure interval. This property should only be called if the `ChartType` is `tsopt.ChartType.PointAndFigure`; otherwise, its value is not meaningful.

The value returned by this property can be either `tsopt.PointFigureBasis.Close` or `tsopt.PointFigureBasis.HighLow`.

`property bool PointFigureOneStepBack { read; }`

Returns true if one-step-back is enabled for a Point-and-Figure interval. This property should only be called if the `ChartType` is `tsopt.ChartType.PointAndFigure`; otherwise, its value is not meaningful.

`property double RenkoBrickSize { read; }`

Returns the brick size for a Renko interval. This property should only be called if the `ChartType` is `tsopt.ChartType.Renko`; otherwise, its value is not meaningful.

`property double RenkoBrickOffset { read; }`

Returns the brick offset for a Renko interval. This property should only be called if the `ChartType` is `tsopt.ChartType.Renko`; otherwise, its value is not meaningful. Note that this property will be non-zero only for a Mean Renko or Spectrum Renko interval.

`string Describe();`

Returns a string that describes the current interval (e.g. "5 min" for a 5 minute bar, "Daily" for a daily bar, or "Kagi 0.5, 1 tick" for a tick-based Kagi bar with a 0.5 reversal size).

History Class

The `History` class defines the history range for an optimization. It provides the following properties and methods.

`property tsopt.Security EndHistory { read; }`

Returns the parent object in the job definition tree, which is a `Security` object. This property is provided in order to support tree-style definitions.

`property tsopt.HistoryRangeType RangeType { read; }`

Returns the type of the history range. This can be one of the following values:

- `tsopt.HistoryRangeType.FirstDate`
- `tsopt.HistoryRangeType.DaysBack`
- `tsopt.HistoryRangeType.WeeksBack`
- `tsopt.HistoryRangeType.MonthsBack`
- `tsopt.HistoryRangeType.YearsBack`
- `tsopt.HistoryRangeType.BarsBack`

`property int RangeSize { read; }`

Returns the size of the range, i.e., the number of units specified by the `RangeType` property.

If `RangeType` is `tsopt.HistoryRangeType.FirstDate`, the `RangeSize` property will return 0.

property DateTime LastDate { read; write; }

Gets or sets the last date of the history range.

property DateTime FirstDate { read; write; }

Gets or sets the first date of the history range.

If the range is currently using a different method, setting this property will force the range to use the First Date method.

This property should only be read if the `RangeType` is `tsopt.HistoryRangeType.FirstDate`; otherwise, its value is not meaningful.

property DateTime LastDateString { read; write; }

Gets or sets the last date of the history range as a string. The string is formatted using the default date format for the current locale.

If a string with an unrecognized date format is assigned to `LastDateString`, the API will throw an `InvalidOperationException` with the message "Invalid DateTime format."

property DateTime FirstDateString { read; write; }

Gets or sets the first date of the history range as a string. The string is formatted using the default date format for the current locale.

If a string with an unrecognized date format is assigned to `FirstDateString`, the API will throw an `InvalidOperationException` with the message "Invalid DateTime format."

If the history range is currently using a different method, setting this property will force the range to use the First Date method.

This property should only be read if the `RangeType` is `tsopt.HistoryRangeType.FirstDate`; otherwise, its value is not meaningful.

property int DaysBack { read; write; }

Gets or sets the number of days back in the history range.

If the range is not currently using Days Back, the property will return zero. However, setting this property will force the range to use the Days Back method.

property int WeeksBack { read; write; }

Gets or sets the number of weeks back in the history range.

If the range is not currently using Weeks Back, the property will return zero. However, setting this property will force the range to use the Weeks Back method.

`property int MonthsBack { read; write; }`

Gets or sets the number of months back in the history range.

If the range is not currently using Months Back, the property will return zero. However, setting this property will force the range to use the Months Back method.

`property int YearsBack { read; write; }`

Gets or sets the number of years back in the history range.

If the range is not currently using Years Back, the property will return zero. However, setting this property will force the range to use the Years Back method.

`property int BarsBack { read; write; }`

Gets or sets the number of bars back in the history range.

If the range is not currently using Bars Back, the property will return zero. However, setting this property will force the range to use the Bars Back method.

`tsopt.History SetLastDate(int year, int month, int day);`

Sets the last date of the history range using the year, month, and day. This method also returns the `History` object, so it can be used in tree-style definitions.

`tsopt.History SetLastDate(DateTime date);`

Sets the last date of the history range using a `DateTime` value. This method also returns the `History` object, so it can be used in tree-style definitions.

`tsopt.History SetLastDate(string date);`

Sets the last date of the history range using a string, which should follow the default date format for the current locale. This method also returns the `History` object, so it can be used in tree-style definitions.

If the string has an unrecognized date format, the API will throw an `InvalidOperationException` with the message "Invalid DateTime format."

`tsopt.History SetFirstDate(int year, int month, int day);`

Sets the first date of the history range using the year, month, and day. This method also returns the `History` object, so it can be used in tree-style definitions.

If the history range is currently specified using a number of units, this method will override that setting.

`tsopt.History SetFirstDate(DateTime date);`

Sets the first date of the history range using a `DateTime` value. This method also returns the `History` object, so it can be used in tree-style definitions.

If the history range is currently specified using a range of units, this method will override that setting.

`tsopt.History SetFirstDate(string date);`

Sets the first date of the history range using a string, which should follow the default date format for the current locale. This method also returns the `History` object, so it can be used in tree-style definitions.

If the string has an unrecognized date format, the API will throw an `InvalidOperationException` with the message "Invalid DateTime format."

If the history range is currently specified using a range of units, this method will override that setting.

`tsopt.History SetDaysBack(int daysBack);`

Sets the number of days back from the last date. This method also returns the `History` object, so it can be used in tree-style definitions.

If the history range is currently specified using a first date or a different unit, this method will override that setting.

`tsopt.History SetWeeksBack(int weeksBack);`

Sets the number of weeks back from the last date. This method also returns the `History` object, so it can be used in tree-style definitions.

If the history range is currently specified using a first date or a different unit, this method will override that setting.

`tsopt.History SetMonthsBack(int monthsBack);`

Sets the number of months back from the last date. This method also returns the `History` object, so it can be used in tree-style definitions.

If the history range is currently specified using a first date or a different unit, this method will override that setting.

`tsopt.History SetYearsBack(int yearsBack);`

Sets the number of years back from the last date. This method also returns the `History` object, so it can be used in tree-style definitions.

If the history range is currently specified using a first date or a different unit, this method will override that setting.

`tsopt.History SetBarsBack(int barsBack);`

Sets the number of bars back from the last date. This method also returns the `History` object, so it can be used in tree-style definitions.

If the history range is currently specified using a first date or a different unit, this method will override that setting.

`string Describe();`

Returns a string that describes the history range, e.g. “Last Date: 12/31/2012, 30 days back”.

OptSymbol Class

The `OptSymbol` class defines an optimization over a list of symbols. The optimizer will test each symbol for the security in combination with any other optimized parameters.

`property tsopt.Security EndOptSymbol { read; }`

Returns the parent object in the job definition tree, which is a `Security` object. This property is provided in order to support tree-style definitions.

`property int Count { read; }`

Returns the number of symbols to optimize.

`property default string Symbol[int] { read; write; }`

Gets or sets the symbol at the specified index. The `OptSymbol` class acts as a collection of symbols, and you can access the symbol at a specific index using array syntax. For example, the following code gets the first symbol to optimize for the first security:

```
sym = job.Securities[0].OptSymbol[0];
```

The following code sets the third symbol to optimize for the first security:

```
job.Securities[0].OptSymbol[2] = "CSCO";
```

Note that a symbol must already be defined at the specified index; otherwise, getting or setting a symbol at that index will throw an `OptimizationException`. Use the `AddSymbol` method to add symbols to the list.

`tsopt.OptSymbol AddSymbol(string symbol);`

Adds a symbol to the list of symbols to optimize.

The method returns the `OptSymbol` object, so you can chain `AddSymbol` methods together to define a list of symbols. For example:

```
job.Securities[0]
   .OptSymbol
      .AddSymbol("AAPL")
      .AddSymbol("CSCO")
      .AddSymbol("MSFT");
```

`tsopt.OptSymbol SetSymbol(int pos, string symbol);`

Sets the symbol at the specified position in the list. This is equivalent to setting a symbol with the indexed property.

The method returns the `OptSymbol` object, so you can chain `SetSymbol` methods together.

`tsopt.OptSymbol DeleteSymbol(int pos);`

Deletes the symbol at the specified position in the list.

`tsopt.OptSymbol Clear();`

Clears the list of symbols to optimize, leaving it empty.

`string Describe();`

Returns a string that describes the optimized symbols. It is formatted as a comma-delimited list, e.g., "AAPL, CSCO, MSFT".

OptInterval Class

The `OptInterval` class defines an optimization over a list of different intervals. The optimizer will test each interval for the security in combination with any other optimized parameters.

`property tsopt.Security EndOptInterval { read; }`

Returns the parent object in the job definition tree, which is a `Security` object. This property is provided in order to support tree-style definitions.

`property int Count { read; }`

Returns the number of intervals to optimize.

`property bool HasAdvancedIntervals { read; }`

Returns true if the list of optimized intervals contains at least one advanced interval.

`property default tsopt.Interval Interval[int] { read; }`

Gets the `Interval` object at the specified index. The `OptInterval` class acts as a collection of intervals, and you can access the interval at a specific index using array syntax. For example, the following code gets the first interval to optimize for the first security:

```
interval = job.Securities[0].OptInterval[0];
```

You can call methods and properties on the returned `Interval` object to modify or query the interval. For example, the following code changes the third interval to Daily:

```
job.Securities[0].OptInterval[2].SetDailyChart();
```

Note that an interval must already be defined at the specified index; otherwise, getting or setting an interval at that index will throw an `OptimizationException`. Use one of the `Add...` methods to add intervals to the list.

`tsopt.Interval AddInterval();`

Adds a default 5 minute interval to the list and returns the `Interval` object.

This method is intended for UI-based client applications. When the user asks to add an interval, the application can use this method to add a default interval. Then the application can provide controls which allow the user to change the interval as desired.

`tsopt.OptInterval DeleteInterval(int pos);`

Deletes the interval at the specified position in the list.

`tsopt.OptInterval Clear();`

Clears the list of optimized intervals, leaving it empty.

`tsopt.OptInterval AddTickChart(int ticks);`

Adds a tick-based interval with the specified number of ticks per bar.

Note: All of the `Add...` methods in the `OptInterval` class return the `OptInterval` object so that you can call additional methods on that object. This allows `Add...` calls to be chained together to define a list of intervals. For example, the following code adds a tick interval, a seconds interval, and a minute interval:

```
job.Securities[0]
   .OptInterval
      .AddTickChart(500)
      .AddSecondChart(15)
      .AddMinuteChart(1);
```

`tsopt.OptInterval AddVolumeChart(int shares);`

Adds a volume-based interval with the specified number of shares per bar.

`tsopt.OptInterval AddSecondChart(int seconds);`

Adds a second-based interval with the specified number of seconds per bar.

`tsopt.OptInterval AddMinuteChart(int minutes);`

Adds a minute-based interval with the specified number of minutes per bar.

`tsopt.OptInterval AddDailyChart();`

Adds a daily interval.

`tsopt.OptInterval AddWeeklyChart();`

Adds a weekly interval.

`tsopt.OptInterval AddMonthlyChart();`

Adds a monthly interval.

```
tsopt.OptInterval AddKagiChart(tsopt.Compression resolutionUnit,
   int resolutionQty, tsopt.KagiReversalMode reversalMode,
   double reversalSize);
```

Adds a Kagi interval based on the underlying time interval specified by `resolutionUnit` and `resolutionQty`. The `reversalMode` and `reversalSize` arguments determine how the bars are built.

The `reversalMode` argument can be either `tsopt.KagiReversalMode.FixedPrice` or `tsopt.KagiReversalMode.Percent`.

```
tsopt.OptInterval AddKaseChart(tsopt.Compression resolutionUnit,
   double targetRange);
```

Adds a Kase interval based on the underlying time interval specified by `resolutionUnit`. The resolution quantity for Kase bars is always 1. The `targetRange` argument determines how the bars are built.

```
tsopt.OptInterval AddLineBreakChart(tsopt.Compression resolutionUnit,
   int resolutionQty, int lineBreaks);
```

Adds a Line Break interval based on the underlying time interval specified by `resolutionUnit` and `resolutionQty`. The `lineBreaks` argument determines how the bars are built.

```
tsopt.OptInterval AddMomentumChart(tsopt.Compression resolutionUnit,
   int resolutionQty, double range);
```

Adds a Momentum interval based on the underlying time interval specified by `resolutionUnit` and `resolutionQty`. The `range` argument determines how the bars are built.

```
tsopt.OptInterval AddPointFigureChart(
   tsopt.Compression resolutionUnit, int resolutionQty,
   double boxSize, int reversal,
   tsopt.PointFigureBasis basis, bool enableOneStepBack);
```

Adds a Point-and-Figure interval based on the underlying time interval specified by `resolutionUnit` and `resolutionQty`. The other arguments determine how the bars are built.

The `basis` argument can be either `tsopt.PointFigureBasis.Close` or `tsopt.PointFigureBasis.HighLow`.

`tsopt.OptInterval AddRangeChart(tsopt.Compression resolutionUnit, int resolutionQty, double range);`

Adds a Range Chart interval based on the underlying time interval specified by `resolutionUnit` and `resolutionQty`. The `range` argument determines how the bars are built.

`tsopt.OptInterval AddRenkoChart(double brickSize);`

Adds a Renko interval, which always uses an underlying interval of 1 tick. The `brickSize` argument determines how the bars are built.

`tsopt.OptInterval AddMeanRenkoChart(double brickSize);`

Adds a Mean Renko interval, which always uses an underlying interval of 1 tick. The `brickSize` argument determines how the bars are built.

`tsopt.OptInterval AddSpectrumRenkoChart(double brickSize, double brickOffset);`

Adds a Spectrum Renko interval, which is equivalent to a Custom Renko interval in Charting. This interval always uses an underlying interval of 1 tick. The `brickSize` and `brickOffset` arguments determine how the bars are built.

`string Describe();`

Returns a string that describes the optimized interval definition. The string is formatted as a list of interval descriptions separated by semicolons (e.g., "30 sec; 1 min; 3 min"). A semicolon is used as the delimiter because some advanced interval descriptions contain a comma.

SecurityOptions Class

The `SecurityOptions` class allows you to get or set the session name, volume usage, bar building, and time zone options for a security. It may be obtained from the `SecurityOptions` property of a `Security` object.

`property tsopt.Security EndSecurityOptions { read; }`

Returns the parent object in the job definition tree, which is a `Security` object. This property is provided in order to support tree-style definitions.

`property string SessionName { read; write; }`

Gets or sets the session name for the security. If the session name is empty, the job definition uses the default session.

property tsopt.ForVolumeUse ForVolumeUse { read; write; }

Gets or sets the volume usage option for the security. It can have one of the following values:

- `tsopt.ForVolumeUse.VolumeAndOI` (volume keywords return volume and open interest)
- `tsopt.ForVolumeUse.Volume` (volume keywords return up and down volume)
- `tsopt.ForVolumeUse.TickCount` (volume keywords return up and down ticks)

property tsopt.BarBuilding BarBuilding { read; write; }

Gets or sets the bar building option for the security. It can have one of the following values:

- `tsopt.BarBuilding.SessionHours`
- `tsopt.BarBuilding.NaturalHours`

property elsystem.TimeZone TimeZone { read; write; }

Gets or sets the time zone for the security. It can have one of the following values:

- `elsystem.TimeZone.Local`
- `elsystem.TimeZone.Exchange`

```
tsopt.SecurityOptions SetSessionName(string sessionName);
tsopt.SecurityOptions SetForVolumeUse(
   tsopt.ForVolumeUse forVolumeUse);
tsopt.SecurityOptions SetBarBuilding(tsopt.BarBuilding barBuilding);
tsopt.SecurityOptions SetTimeZone(elsystem.TimeZone timeZone);
```

These methods perform the same operation as setting the equivalent property. However, they also return the `SecurityOptions` object so that you can chain the `Set…` methods together. This is useful when specifying options within a tree-style job definition.

Strategies Class

The `Strategies` class defines the strategies to use for an optimization. Every job definition must include at least one strategy. When multiple strategies are included in a job, they all work together to form a master strategy.

property tsopt.Job EndStrategies { read; }

Returns the parent object in the job definition tree, which is a `Job` object. This property is provided in order to support tree-style definitions.

property int Count { read; }

Returns the number of strategies in the definition.

property default tsopt.Strategy Strategy[int] { read; }

Returns the `Strategy` object at the specified index. The `Strategies` object acts as a collection of `Strategy` objects, and you can access the strategy at a specific index using array syntax. For example, the following code accesses the first strategy in the job definition:

```
strategy = job.Strategies[0];
```

tsopt.Strategy AddStrategy(string strategyName);

Adds the strategy with the specified name to the job definition and returns the corresponding `Strategy` object. You can then call methods on that object to define the properties of the strategy.

tsopt.Strategy AddStrategy(string strategyName, bool populateInputs);

Adds the strategy with the specified name to the job definition and returns the corresponding `Strategy` object. If `populateInputs` is true, this method will also add all of the strategy's inputs to the definition, using the default value for each input. This is useful for UI-based applications that need to display the available inputs to the user. An application can use this method to add the strategy and its inputs; then it can query the `Strategy` object to get the inputs and display them to the user.

Note: Populating the `Strategy` object with the default inputs does not prevent any of those inputs from being changed or optimized. An application can later call one of the methods in the `ELInputs` object to optimize any input or to set it to a different value.

tsopt.Strategies DeleteStrategy(int pos);

Deletes the strategy at the specified position. Note that this only deletes the strategy from the job definition. It does **not** delete the strategy from the TradeStation environment.

Strategy Class

The `Strategy` class defines a single strategy in an optimization. It provides the following methods and properties.

property tsopt.Strategies EndStrategy { read; }

Returns the parent object in the job definition tree, which is a `Strategies` object. This property is provided in order to support tree-style definitions.

property string Name { read; }

Returns the name of the strategy.

Note that you cannot change the name of a strategy in a job definition. If you wish to replace one strategy with another in a job definition, you must delete the old strategy and add the new one.

property tsopt.ELInputs ELInputs { read; }

Returns an `ELInputs` object that defines the inputs for the strategy. This includes both fixed inputs and optimized inputs. You can call methods on the `ELInputs` object to set or optimize the inputs.

property bool StrategyEnabled { read; write; }

Gets or sets the strategy's enabled status (i.e., whether it participates in the optimization).

tsopt.Strategy SetStrategyEnabled(bool enabled);

Sets the strategy's enabled status. This is equivalent to setting the `StrategyEnabled` property, but this method can be used in tree-style job definitions.

tsopt.Strategy OptStrategyEnabled();

Optimizes the strategy's enabled status. The optimizer will run tests with the strategy enabled and disabled, in combination with any other optimized parameters. This can help you determine whether the strategy makes a useful contribution to a group of cooperating strategies.

This method should be used only when multiple strateges are defined for a job.

property tsopt.SignalStates SignalStates { read; }

Returns a `SignalStates` object which can be used to get or set signal states for the strategy. These determine whether the buy, sell, short, and cover signals are enabled or disabled.

See the description of the `SignalStates` class in this *API Reference* for more information.

property tsopt.IntrabarOrders IntrabarOrders { read; }

Returns an `IntrabarOrders` object which can be used to get or set intrabar order options for the strategy.

See the description of the `IntrabarOrders` class in this *API Reference* for more information.

ELInputs Class

The `ELInputs` class is used to set or optimize the inputs for a strategy. You can also use this class to obtain information about the inputs in a strategy definition.

property tsopt.Strategy EndELInputs { read; }

Returns the parent object in the job definition tree, which is a `Strategy` object. This property is provided in order to support tree-style definitions.

property int Count { read; }

Returns the number of inputs in this strategy definition.

Note that the `Count` property does not necessarily return the total number of inputs for the strategy as it is defined in EasyLanguage. Instead, the `Count` property returns the number of inputs that have been specified for the strategy *in this job definition*. A job definition is not required to specify every input for a strategy: it can include only those inputs which are optimized or which have non-default values. Thus, `Count` may return a number lower than the total input count for the strategy.

If you pass true for the `populateInputs` argument when you add a strategy to the job definition, then all of the inputs for that strategy will be automatically included. In that case, the `Count` property will return the total input count for the strategy.

`property default tsopt.ELInput ELInput[int] { read; }`

Returns the `ELInput` object for the specified index. The `ELInputs` object acts as a collection of `ELInput` objects, and you can access the input at a specific index using array syntax. For example, the following code gets the first input for the first strategy:

```
input = job.Strategies[0].ELInputs[0];
```

The main reason to get an `ELInput` object is to find out the properties of the input. To set or optimize an input, you must use other methods from the `ELInputs` class, as described in this section.

Note that an input must already be defined at the specified index; otherwise, accessing that index will throw an `OptimizationException`. If you pass true for the `populateInputs` argument when you add a strategy, it will automatically add all of the strategy's inputs to the definition. Otherwise, the `ELInputs` object will contain only those inputs that you explicitly define with either the `SetInput` method or one of the `Opt…` methods in the `ELInputs` class.

`tsopt.ELInputs SetInput(string name, string value);`

Sets the value for the input with the specified name. It is not necessary to call the `SetInput` method if you wish to use the default value for an input; however, you can use this method to specify a different value.

This version of the `SetInput` method assumes that the input is numeric (the most common case). Note that the value is passed as a string. This allows the input value to be a reserved word (such as "Close") or a numeric expression (such as "(High + Low) / 2"). To specify a number, you can pass a string representation of that number, or you can use the version of `SetInput` that accepts the value as a double (see below).

If the `name` argument does not specify a valid input for the strategy, you will receive a validation error when you start the optimization.

`tsopt.ELInputs SetInput(string name, string value,`
`tsopt.InputType type);`

Sets the value for the input with the specified name. The `type` argument indicates the type of the input. Use this version of `SetInput` if the input has a non-numeric type.

The `type` argument can have one of the following values:

- `tsopt.InputType.NumericType`
- `tsopt.InputType.TrueFalseType`
- `tsopt.InputType.TextType`

If you want to set a text input to a literal text value, the `value` argument of `SetInput` needs to include an opening and closing quote. (In other words, the quotes must be part of the value itself.) There are a couple ways to handle this:

- If you are getting the input value from a user interface (e.g. a `TextBox` object), make sure the user knows to type quote marks at the beginning and end of a literal text value. For example, this is how literal text inputs are entered in the TradeStation platform.
- If you are specifying a text value directly in your code, you can use the `SetInputAsText` method to set the value. That method automatically wraps the value in quotes for you, since this is slightly cumbersome to do in EasyLanguage. You may also want to use `SetInputAsText` if the text value coming from a user interface is *always* text and you don't want to require the user to enclose it in quotes.

`tsopt.ELInputs SetInput(string name, double value);`

Sets the value of a numeric input with the specified name. This version allows you to pass a numeric value directly. Note that it is also fine to pass the string representation of a number (when using the first version of `SetInput`), since this is just treated as an expression that evaluates to that number. The following calls are equivalent:

- `strategy.ELInputs.SetInput("Length", 24);`
- `strategy.ELInputs.SetInput("Length", "24");`

`tsopt.ELInputs SetInputAsText(string name, string value);`

Sets the value of a text input with the specified name. When you use this method, you don't need to embed quotes at the beginning and end of the text value – the method does it for you automatically.

The following calls are equivalent:

- `Strategy.ELInputs.SetInputAsText("Greeting", "Hello");`
- ```
  Strategy.ELInputs.SetInput("Greeting",
     DoubleQuote + "Hello" + DoubleQuote, tsopt.InputType.TextType);
  ```

`tsopt.ELInputs OptBool(string name);`

Optimizes the Boolean (True/False) input with the specified name. The optimizer will test both true and false values for the input.

Note that this method merely specifies that the input should be optimized. The actual optimization occurs when you call the `Optimizer.StartJob` method.
  ```

```
tsopt.ELInputs OptRange(string name, double start, double stop,
   double step);
```

Optimizes the input with the specified name over a range of values. The optimizer will test each value by beginning with the `start` value and incrementing the value by `step` until it reaches the `stop` value.

Note that this method merely specifies that the input should be optimized. The actual optimization occurs when you call the `Optimizer.StartJob` method.

```
tsopt.OptList OptList(string name);
```

Optimizes the input with the specified name over a list of values. The optimizer will test each value in the list on the input.

The method returns an `OptList` object that represents the list of values. You can call the `AddValue` method on that object to add values to the list.

If the specified input is already optimized by list, the `OptList` method will return the existing `OptList` object. You may then call methods on that object to modify the existing list of values.

This version of the `OptList` method assumes that the input is numeric. To optimize a non-numeric input, call the version of `OptList` that accepts an `InputType`.

```
tsopt.OptList OptList(string name, tsopt.InputType type);
```

Optimizes the input with the specified name over a list of values. The `type` argument indicates the type of the input. Use this version of `OptList` if the input has a non-numeric type.

See the preceding version of `OptList` for a more detailed description of this method.

ELInput Class

The `ELInput` class represents a single strategy input in a job definition. This is strictly an informational class: it is provided so that a client application can learn the properties of an input. To set or optimize an input, you must use one of the methods in the `ELInputs` class.

An `ELInput` object can be obtained by indexing into an `ELInputs` object.

```
property string Name { read; }
```

Returns the name of the input.

```
property string Value { read; }
```

Returns a string representation of the input's value. This may be a number such as "15", a reserved word such as "Close", or an expression such as "(High + Low) / 2". If the value is a text literal, it will contain embedded quotes at the beginning and end of the string. If the value is Boolean, it will be "true" or "false" (or some variant of these – Boolean values are not case-sensitive).

If the input is optimized, the `Value` property returns an empty string. You can use the `InputMode` property to determine whether the input is optimized, and if so, what kind of optimization it uses.

`property tsopt.InputType Type { read; }`

Returns the data type of the input as one of the following values:

- `tsopt.InputType.NumericType`
- `tsopt.InputType.TextType`
- `tsopt.InputType.TrueFalseType`

`property tsopt.InputMode InputMode { read; }`

Returns a `tsopt.InputMode` value that indicates how the input is evaluated. You can use this value to determine whether the input is optimized or fixed. If it is optimized, you can determine what kind of optimization it uses.

This property can return one of the following values:

- `tsopt.InputMode.FixedInput` (for a fixed input)
- `tsopt.InputMode.OptRange` (optimized over a range)
- `tsopt.InputMode.OptList` (optimized over a list)
- `tsopt.InputMode.OptBool` (optimized Boolean input)

`property tsopt.OptRange AsOptRange { read; }`

Returns the input as an `OptRange` object that represents an optimization over a range. You can then use this object to get or set the properties of the range.

You should access this property **only** if the `InputMode` property returns `tsopt.InputMode.OptRange`. Otherwise, accessing this property will throw an `OptimizationException`.

For example, the following code doubles the step value of an input **if** it is optimized by range. (This code assumes you have declared a local variable `optRange` of type `tsopt.OptRange`.)

```
if input.InputMode = tsopt.InputMode.OptRange then begin
   optRange = input.AsOptRange;
   optRange.Step = optRange.Step * 2;
end;
```

`property tsopt.OptList AsOptList { read; }`

Returns the input as an `OptList` object that represents an optimization over a list. You can then use this object to modify or read the list of values to optimize.

You should access this property **only** if the `InputMode` property returns `tsopt.InputMode.OptList`. Otherwise, accessing this property will throw an `OptimizationException`.

For example, the following code deletes the first value from the list of values to optimize, but only if the input is optimized by list and has more than one value. (This code assumes you have declared a local variable `optList` of type `tsopt.OptList`.)

```
if input.InputMode = tsopt.InputMode.OptList then begin
   optList = input.AsOptList;
   if optList.Count > 1 then
      optList.DeleteValue(0);
end;
```

`string Describe();`

Returns a string that describes the input.

For fixed inputs, the description will include the input name and value, e.g., "Length: 20".

For optimized inputs, the description will include the input name and the optimization parameters, e.g. "Length: Optimize: 10..30:1" for an optimized range.

OptRange Class

The `OptRange` class represents a strategy input that is optimized over a range. You can use this class to get or set the optimized range properties for the input.

To obtain an `OptRange` object, you must access the `AsOptRange` property on an input that is *already* optimized by range. To optimize by range on an input that is currently fixed or optimized by another method, get an `ELInputs` object for the strategy and call the `OptRange` method on that object, passing the name of the input.

`property string Name { read; }`

Returns the name of the optimized input.

`property double Start { read; write; }`

Gets or sets the start value of the optimized range.

`property double Stop { read; write; }`

Gets or sets the stop value of the optimized range.

`property double Step { read; write; }`

Gets or sets the step value of the optimized range.

`tsopt.OptRange SetStart(double start);`

Sets the start value of an optimized range. This method can be used in a tree-style definition.

`tsopt.OptRange SetStop(double stop);`

Sets the stop value of an optimized range. This method can be used in a tree-style definition.

`tsopt.OptRange SetStep(double step);`

Sets the step value of an optimized range. This method can be used in a tree-style definition.

`string Describe();`

Returns a string that describes the optimized input, e.g., "Length: Optimize: 10..30:1".

OptList Class

The `OptList` class represents a strategy input that is optimized over a list of values. You can use this class to modify or read the values in the list.

There are two ways to obtain an OptList object:

- Call the `OptList` method on an `ELInputs` object, passing the name of the input to optimize. This returns an `OptList` object that you can use to define the values. If the input was already optimized by list, the `OptList` object will contain the current set of values to optimize. Otherwise, it will be empty, and you will need to call `AddValue` on the object to add one or more values.
- If an input is *already* optimized by list, you can access the `AsOptList` property on its `ELInput` object. This will return an `OptList` object that represents the current list of values.

The input values for an `OptList` follow the same rules as a single input value. See the `SetInput` method in the `ELInputs` class for more information about specifying input values.

The following code optimizes the "BollingerPrice" input over a list. It will test the Close, High, and Low reserved words for the input:

```
job.Strategies[0]
   .OptList("BollingerPrice")
      .AddValue("Close")
      .AddValue("High")
      .AddValue("Low");
```

The `OptList` class provides the following methods and properties.

`property tsopt.ELInputs EndOptList { read; }`

Returns the parent object in the job definition tree, which is an `ELInputs` object. This property is provided in order to support tree-style definitions.

`property string Name { read; }`

Returns the name of the optimized input.

`property int Count { read; }`

Returns the number of values in the list.

property string Item[int] { read; write; }

Gets or sets the value at a specified index in the list using array syntax. The `OptList` object represents a collection of values, and you can access a specific values with array syntax. For example:

```
optList = job.Strategies[0].ELInputs.OptList("BollingerPrice");
first = optList[0];
second = optList[1];
```

A value must already be defined at the specified position. Otherwise, this property will throw an `OptimizationException`. You can use one of the `AddValue` methods to add values to the list.

tsopt.OptList AddValue(string value);

Adds a value to the end of the list.

In a tree-style job definition, multiple `AddValue` calls can be chained together to define the contents of the list, as shown in the example above.

When adding values to the list in a loop, first save the `OptList` object in a local variable. Then you can call the `AddValue` method on that variable within the loop. For example, the following code optimizes the "BollingerPrice" input using the values in a `Vector`.

```
optList = job.Strategies[0].ELInputs.OptList("BollingerPrice");
for j = 0 to values.Count - 1 begin
   optList.AddValue(values[j] astype string);
```

tsopt.OptList AddValue(double value);

Adds a numeric value to the end of the list. Thie method allows you to pass a number directly.

tsopt.OptList AddValueAsText(string value);

Adds a text value to the end of the list. This method automatically adds an embedded quote at the beginning and end of the value string, so that the optimizer will know that it is a literal text value rather than an expression.

tsopt.OptList SetValue(int pos, string value);

Sets the value at the specified position in the list. This is equivalent to setting the indexed property. However, the `SetValue` method also returns the `OptList` object, so you can chain `SetValue` methods together.

tsopt.OptList SetValue(int pos, double value);

Sets a numeric value at the specified position in the list. This method allows you to pass a number directly.

`tsopt.OptList SetValueAsText(int pos, double value);`

Sets a text value at the specified position in the list. This method automatically adds an embedded quote at the beginning and end of the value string, so that the optimizer will know that it is a literal text value rather than an expression.

`tsopt.OptList DeleteValue(int pos);`

Deletes the value at the specified position in the list.

`tsopt.OptList Clear();`

Clears the list of values, leaving it empty.

`string Describe();`

Returns a string that describes the optimized input, e.g., "BollingerPrice: Optimize: Close, High, Low".

SignalStates Class

The `SignalStates` class allows you to get or set individual signal states for a strategy. You can obtain a `SignalStates` object by accessing the `SignalStates` property on a `Strategy` object.

Setting a signal state allows you to enable or disable each type of signal (Buy, Sell, Short, or Cover). You can also specify that Buy or Short signals should be used only to exit trades by specifying the `ExitOnly` state for those signals.

The following signal state values are allowed:

- `tsopt.SignalState.Off`
- `tsopt.SignalState.On`
- `tsopt.SignalState.ExitOnly` (valid only for Buy or Short)

The `SignalStates` class provides the following methods and properties.

`property tsopt.Strategy EndSignalStates { read; }`

Returns the parent object in the job definition tree, which is a `Strategy` object. This property is provided in order to support tree-style definitions.

`property tsopt.SignalState BuyState { read; write; }`
`property tsopt.SignalState SellState { read; write; }`
`property tsopt.SignalState ShortState { read; write; }`
`property tsopt.SignalState CoverState { read; write; }`

Each of these properties gets or sets the signal state for the specified signal type.

For example, the following code turns off Short signals for the first strategy:

```
job.Strategies[0].SignalStates.ShortState = tsopt.SignalState.Off;
```

```
tsopt.SignalStates SetBuyState(tsopt.SignalState state);
tsopt.SignalStates SetSellState(tsopt.SignalState state);
tsopt.SignalStates SetShortState(tsopt.SignalState state);
tsopt.SignalStates SetCoverState(tsopt.SignalState state);
```

Each of these methods sets the signal state for the specified signal type. They can be chained together in a job definition.

For example, the following code turns off both the Sell and Cover signals for the first strategy, thus ensuring that it is always in the market:

```
job.Strategies[0]
   .SignalStates
      .SetSellState(tsopt.SignalState.Off)
      .SetCoverState(tsopt.SignalState.Off);
```

IntrabarOrders Class

The `IntrabarOrders` class allows you to get or set the intrabar order generation options for a strategy. You can obtain an `IntrabarOrders` object by accessing the `IntrabarOrders` property on a `Strategy` object.

The `IntrabarOrders` class provides the following methods and properties.

```
property tsopt.Strategy EndIntrabarOrders { read; }
```

Returns the parent object in the job definition tree, which is a `Strategy` object. This property is provided in order to support tree-style definitions.

```
property bool IntrabarEnabled { read; write; }
```

Gets or sets whether intrabar order generation is enabled.

```
property tsopt.IntrabarOrderMode IntrabarEntryMode { read; write; }
```

Gets or sets the intrabar order entry mode. This can be one of the following values:

- `tsopt.IntrabarOrderMode.Once`
- `tsopt.IntrabarOrderMode.OncePerSignal`
- `tsopt.IntrabarOrderMode.Unlimited`

Note that the `Unlimited` option is not truly unlimited for intrabar entries. The number of intrabar entries will not exceed the value specified by the `IntrabarMaxEntries` property.

```
property tsopt.IntrabarOrderMode IntrabarExitMode { read; write; }
```

Gets or sets the intrabar order exit mode. The possible values are the same as for the entry mode.

```
property int IntrabarMaxEntries { read; write; }
```

Gets or sets the maximum number of intrabar entries per bar. This option is used only if the `IntrabarEntryMode` is `tsopt.IntrabarOrderMode.Unlimited`.

```
tsopt.IntrabarOrders SetIntrabarEnabled(bool enabled);
tsopt.IntrabarOrders SetIntrabarEntryMode(
   tsopt.IntrabarOrderMode mode);
tsopt.IntrabarOrders SetIntrabarExitMode(
   tsopt.IntrabarOrderMode mode);
tsopt.IntrabarOrders SetIntrabarMaxEntries(int maxEntries);
```

Each of these methods sets the specified intrabar order option. Since they return the `IntrabarOrders` object, they can be chained together in a job definition. For example, the following code changes the maximum number of entries per bar from once to 10 entries per bar:

```
job.Strategies[0]
   .IntrabarOrders
      .SetIntrabarEntryMode(tsopt.IntrabarOrderMode.Unlimited)
      .SetIntrabarMaxEntries(10);
```

Settings Class

The `Settings` class defines the global settings for an optimization. These can be settings that apply to all of the strategies in an optimization (e.g. MaxBarsBack), or they can be settings that apply to the optimization itself (e.g. genetic options).

The settings are grouped into the following categories:

- Genetic options (for genetic optimizations)
- Result options (number of tests to keep, fitness function, etc.)
- General options (base currency, MaxBarsBack, Look-Inside-Bar, etc.)
- Costs and Capital options (commission, slippage, capital, etc.)
- Position options (pyramiding options and maximum shares per position)
- Trade Size options (shares or currency per trade)
- Back-Testing options (fill options, market slippage, etc.)
- Out-of-Sample options (size of the out-of-sample range, if any)
- Walk-Forward options (enable/disable walk-forward, name of walk-forward test)

You can obtain a `Settings` object from the `Settings` property on a `Job` object. Then you can access one of its sub-options properties in order to modify settings in a particular category.

See *Specifying Job Settings* in the initial part of this guide for a detailed explanation and examples.

The `Settings` class provides the following methods and properties.

```
property tsopt.Job EndSettings { read; }
```

Returns the parent object in the job definition tree, which is a `Job` object. This property is provided in order to support tree-style definitions.

```
property tsopt.GeneticOptions  GeneticOptions { read; }
property tsopt.ResultOptions   ResultOptions { read; }
property tsopt.GeneralOptions  GeneralOptions { read; }
property tsopt.CostsAndCapital CostsAndCapital { read; }
property tsopt.PositionOptions PositionOptions { read; }
property tsopt.TradeSize       TradeSize { read; }
property tsopt.BackTesting     BackTesting { read; }
property tsopt.OutSample       OutSample { read; }
property tsopt.WalkForward     WalkForward { read; }
```

Each of these properties returns an options object that represents a related group of settings. You can then access the properties of the object to get or set individual options.

The following sections describe the different options classes.

GeneticOptions Class

The GeneticOptions class defines the options for a genetic optimization. These options have an effect only if the job definition specifies a genetic optimization (via the `OptimizationMethod` property in the `Job` class).

This class has the following members:

```
property tsopt.Settings EndGeneticOptions { read; }
```

Returns the parent object in the job definition tree, which is a `Settings` object. This property is provided in order to support tree-style definitions.

```
property int Generations { read; write; }
```

Gets or sets the number of generations for a genetic optimization.

```
property int PopulationSize { read; write; }
```

Gets or sets the population size for a genetic optimization.

```
property double CrossoverRate { read; write; }
```

Gets or sets the crossover rate for a genetic optimization.

```
property double MutationRate { read; write; }
```

Gets or sets the mutation rate for a genetic optimization.

```
property int StressTestSize { read; write; }
```

Gets or sets the stress test size for a genetic optimization. Note that the stress test size must be an integer from 1 to 5. Set the size to 1 if you don’t wish to stress test the optimization.

```
property double StressIncrement { read; write; }
```

Gets or sets the stress test increment for a genetic optimization. This option has an effect only if the stress test size is greater than one.

```
property int EarlyExitGenerations { read; write; }
```

Gets or sets the number of early exit generations for a genetic optimization. If the population fitness does not improve after the specified number of generations, the optimization will exit.

To prevent early exit, set the value of `EarlyExitGenerations` to zero. (This is the default value of the property.)

```
property bool AutoAdjustBasedOnJob { read; write; }
```

Gets or sets a flag that determines whether the genetic options will be automatically adjusted based on the current job definition.

If this property is true, the optimizer will automatically adjust the genetic options based on the number of iterations in the optimization job. This is equivalent to clicking the "Suggest" button when defining the genetic options for an optimization in Charting. In general, a larger number of iterations may cause the optimizer to choose a higher population size and/or generation count. The mutation rate will also be adjusted.

If this property is false, the optimizer will use the genetic options that you specify (or the default values for options that you do not specify).

```
GeneticOptions SetGenerations(int generations);
GeneticOptions SetPopulationSize(int populationSize);
GeneticOptions SetCrossoverRate(double crossoverRate);
GeneticOptions SetMutationRate(double mutationRate);
GeneticOptions SetStressTestSize(int stressTestSize);
GeneticOptions SetStressIncrement(double stressIncrement);
GeneticOptions SetEarlyExitGenerations(int earlyExitGenerations);
GeneticOptions SetAutoAdjustBasedOnJob(bool autoAdjustBasedOnJob);
```

These methods perform the same operation as setting the equivalent property. However, they also return the `GeneticOptions` object so that you can chain the `Set`… methods together. This is useful when specifying options within a tree-style job definition.

ResultOptions Class

The `ResultOptions` class specifies the result options for an optimization. It has the following members:

```
property tsopt.Settings EndResultOptions { read; }
```

Returns the parent object in the job definition tree, which is a `Settings` object. This property is provided in order to support tree-style definitions.

property int NumTestsToKeep { read; write; }

Gets or sets the number of tests to keep in the `Results` object.

Note that this is just the requested number of tests. The actual number of tests may be lower if the optimization produces fewer combinations, or it may be higher if there are tests at the bottom of the results set with tied fitness values. Always use the `Results.TestCount` property to get the actual number of tests.

property tsopt.KeepType KeepType { read; write; }

Gets or sets the keep type, which specifies whether to keep the tests with the highest fitness values (`tsopt.KeepType.Highest`) or the lowest fitness values (`tsopt.KeepType.Lowest`).

property tsopt.MetricID FitnessMetric { read; write; }

Gets or sets the metric to use as the fitness function. For example, you can set this property to `tsopt.MetricID.ProfitFactor` to use Profit Factor as the fitness function.

property tsopt.TradeType FitnessTradeType { read; write; }

Gets or sets the fitness trade type, which specifies what trade type to use when evaluating the fitness function. It can have one of the following values:

- `tsopt.TradeType.AllTrades`
- `tsopt.TradeType.LongTrades`
- `tsopt.TradeType.ShortTrades`

```
ResultOptions SetNumTestsToKeep(int numTests);
ResultOptions SetKeepType(tsopt.KeepType keepType);
ResultOptions SetFitnessMetric(tsopt.MetricID metricID);
ResultOptions SetFitnessTradeType(tsopt.TradeType tradeType);
```

These methods perform the same operation as setting the equivalent property. However, they also return the `ResultOptions` object so that you can chain the `Set…` methods together. This is useful when specifying options within a tree-style job definition.

GeneralOptions Class

The `GeneralOptions` class specifies some general options for an optimization that don't fall neatly into other categories. It has the following members.

property tsopt.Settings EndGeneralOptions { read; }

Returns the parent object in the job definition tree, which is a `Settings` object. This property is provided in order to support tree-style definitions.

property tsopt.BaseCurrency BaseCurrency { read; write; }

Gets or sets the base currency to use for the optimization. You can specify one of the following options:

- `tsopt.BaseCurrency.Symbol`
- `tsopt.BaseCurrency.Account`

property int MaxBarsBack { read; write; }

Gets or sets the MaxBarsBack value for the optimization. This is the maximum bars of history required before the strategy can begin its calculations.

property int BounceRatio { read; write; }

Gets or sets the percentage increment for Bouncing Ticks™.

property bool LookInsideBarEnabled { read; write; }

Gets or sets whether Look-Inside-Bar back-testing is enabled.

property tsopt.Compression LookInsideBarResUnit { read; write; }

Gets or sets the unit to use if Look-Inside-Bar is enabled (ticks, seconds, minutes, etc.). The unit and quantity determine the Look-Inside-Bar resolution. For example, the following code uses 50 ticks for the resolution:

```
generalOptions = job.Settings.GeneralOptions;
generalOptions.LookInsideBarEnabled = true;
generalOptions.LookInsideBarResUnit = tsopt.Compression.Tick;
generalOptions.LookInsideBarResQty  = 50;
```

property int LookInsideBarResQty { read; write; }

Gets or sets the number of units to use for the Look-Inside-Bar resolution.

```
GeneralOptions SetBaseCurrency(tsopt.BaseCurrency baseCurrency);
GeneralOptions SetMaxBarsBack(int maxBarsBack);
GeneralOptions SetBounceRatio(int bounceRatio);
GeneralOptions SetLookInsideBarEnabled(bool enabled);
GeneralOptions SetLookInsideBarResUnit(
   tsopt.Compression resolutionUnit);
GeneralOptions SetLookInsideBarResQty(int resolutionQty);
```

These methods perform the same operation as setting the equivalent property. However, they also return the `GeneralOptions` object so that you can chain the `Set…` methods together. This is useful when specifying options within a tree-style job definition.

CostsAndCapital Class

The `CostsAndCapital` class specifies the cost and capital options for an optimization. It has the following members:

property tsopt.Settings EndCostsAndCapital { read; }

Returns the parent object in the job definition tree, which is a `Settings` object. This property is provided in order to support tree-style definitions.

property tsopt.CommissionMode CommissionMode { read; write; }

Gets or sets the commission mode, which determines how the commission amount and/or percent is interpreted. It can have the following values:

- `tsopt.CommissionMode.FixedPerShare` (amount per share)
- `tsopt.CommissionMode.FixedPerTrade` (amount per trade)
- `tsopt.CommissionMode.Percent` (% of total cost)
- `tsopt.CommissionMode.PerTradePlusPct` (amount + % total cost)
- `tsopt.CommissionMode.PercentWithMin` (% total cost with min amount)

property double CommissionAmount { read; write; }

Gets or sets the commission amount. The commission mode determines how this amount is used when calculating the commission. This property is used only with the following modes: `FixedPerShare`, `FixedPerTrade`, `PerTradePlusPct`, `PercentWithMin`.

property double CommissionPercent { read; write; }

Gets or sets the commission percent. The commission mode determines how this percentage is used when calculating the commission. This property is used only with the following modes: `Percent`, `PerTradePlusPct`, or `PercentWithMin`.

property tsopt.SlippageMode SlippageMode { read; write; }

Gets or sets the slippage mode, which determines how the slippage amount is interpreted. It can have the following values:

- `tsopt.SlippageMode.FixedPerShare`
- `tsopt.SlippageMode.FixedPerTrade`

property double SlippageAmount { read; write; }

Gets or sets the slippage amount. The slippage mode determines whether this is interpreted as the amount per share or the amount per trade.

property double InterestRate { read; write; }

Gets or sets the interest rate to use when calculating performance metrics.

property double Capital { read; write; }

Gets or sets the capital to use when calculating performance metrics.

```
property tsopt.CommissionBySymbol CommissionBySymbol { read; }
```

Returns a `CommissionBySymbol` object that allows you to specify a different commission for each symbol in an optimization. This is useful when optimizating over symbols that have different commission rates.

See the *CommissionBySymbol Class* entry below for information and examples about specifying commission by symbol.

```
property tsopt.SlippageBySymbol SlippageBySymbol { read; }
```

Returns a `SlippageBySymbol` object that allows you to specify a different slippage amount for each symbol in an optimization. This is useful when optimizing over symbols that typically have different slippage values.

See the *SlippageBySymbol Class* entry below for information and examples about specifying slippage by symbol.

```
CostsAndCapital SetCommissionMode(
   tsopt.CommissionMode commissionMode);
CostsAndCapital SetCommissionAmount(double commissionAmount);
CostsAndCapital SetCommissionPercent(double commissionPercent);
CostsAndCapital SetSlippageMode(tsopt.SlippageMode slippageMode);
CostsAndCapital SetSlippageAmount(double slippageAmount);
CostsAndCapital SetInterestRate(double interestRate);
CostsAndCapital SetCapital(double capital);
```

These methods perform the same operation as setting the equivalent property. However, they also return the `CostsAndCapital` object so that you can chain the `Set...` methods together. This is useful when specifying options within a tree-style job definition.

PositionOptions Class

The `PositionOptions` class specifies the position options for an optimization. It has the following members:

```
property tsopt.Settings EndPositionOptions { read; }
```

Returns the parent object in the job definition tree, which is a `Settings` object. This property is provided in order to support tree-style definitions.

```
property tsopt.PyramidingMode PyramidingMode { read; write; }
```

Gets or sets the pyramiding mode, which determines whether pyramiding is enabled, and if so, what kind of pyramiding to use. It can have one of the following values:

- `tsopt.PyramidingMode.None` (disables pyramiding)
- `tsopt.PyramidingMode.DifEntry` (allows pyramiding for different entry signals)
- `tsopt.PyramidingMode.AnyEntry` (allows pyramiding for any entry signal)

property int MaxPyramidingEntries { read; write; }

Gets or sets the maximum number of pyramiding entries.

property int MaxSharesPerPosition { read; write; }

Gets or sets the maximum shares or contracts per position.

```
PositionOptions SetPyramidingMode(tsopt.PyramidingMode mode);
PositionOptions SetMaxPyramidingEntries(int maxPyramidingEntries);
PositionOptions SetMaxSharesPerPosition(int maxSharesPerPosition);
```

These methods perform the same operation as setting the equivalent property. However, they also return the `PositionOptions` object so that you can chain the `Set...` methods together. This is useful when specifying options within a tree-style job definition.

TradeSize Class

The `TradeSize` class specifies the trade size options for an optimization. It has the following members:

property tsopt.Settings EndTradeSize { read; }

Returns the parent object in the job definition tree, which is a `Settings` object. This property is provided in order to support tree-style definitions.

property tsopt.TradeSizeMode TradeSizeMode { read; write; }

Gets or sets the trade size mode, which determines whether to use shares or currency to set the trade size. It can have one of the following values:

- `tsopt.TradeSizeMode.FixedShares`
- `tsopt.TradeSizeMode.FixedCurrency`

property int FixedShares { read; write; }

Gets or sets the number of shares to use for trades. This value is used only if the trade size mode is set to `tsopt.TradeSizeMode.FixedShares`.

property double FixedCurrency { read; write; }

Gets or sets the amount of currency to use per trade. This value is used only if the trade size mode is set to `tsopt.TradeSizeMode.FixedCurrency`.

property int MinShares { read; write; }

Gets or sets the minimum number of shares per trade. This option is used only if the trade size mode is set to `tsopt.TradeSizeMode.FixedCurrency`.

property int RoundDownShares { read; write; }

Gets or sets the round-down shares value. The number of shares will be rounded down to the nearest multiple of this value. This option is used only if the trade size mode is set to `tsopt.TradeSizeMode.FixedCurrency`.

```
TradeSize SetTradeSizeMode(tsopt.TradeSizeMode tradeSizeMode);
TradeSize SetFixedShares(int fixedShares);
TradeSize SetFixedCurrency(double fixedCurrency);
TradeSize SetMinShares(int minShares);
TradeSize SetRoundDownShares(int roundDownShares);
```

These methods perform the same operation as setting the equivalent property. However, they also return the `TradeSize` object so that you can chain the `Set...` methods together. This is useful when specifying options within a tree-style job definition.

BackTesting Class

The `BackTesting` class specifies the back-testing options for an optimization. It has the following members:

property tsopt.Settings EndBackTesting { read; }

Returns the parent object in the job definition tree, which is a `Settings` object. This property is provided in order to support tree-style definitions.

property tsopt.FillModel FillModel { read; write; }

Gets or sets the fill model, which determines how limit orders will be filled during backtesting. It can have one of the following values:

- `tsopt.FillModel.PriceAtLimit` (Fill entire order when trade occurs at limit price or better.)
- `tsopt.FillModel.PriceExceedsLimit` (Fill entire order when trade price exceeds limit price.)
- `tsopt.FillModel.VolumeBased` (Fill order at limit price or better after `FillVolume` shares have traded at limit price or better.)
- `tsopt.FillModel.TradeBased` (Fill order at limit price or better after `FillTrades` trades occur at limit price or better.)

property int FillVolume { read; write; }

Gets or sets the volume that must be traded before a limit order is filled. This option is only used if the `FillModel` is `tsopt.FillModel.VolumeBased`.

property int FillTrades { read; write; }

Gets or sets the number of trades that must occur before a limit order is filled. This option is used only if the `FillModel` is `tsopt.FillModel.TradeBased`.

property double MarketSlippage { read; write; }

Gets or sets the amount of slippage to apply to market orders.

```
property bool PriceInsideBar { read; write; }
```

Gets or sets the price-inside-bar setting, which indicates whether the market fill price can be outside the bar high/low. Set this property to true if you wish to keep the fill price inside the bar.

```
property bool OptimizeIOG { read; write; }
```

Gets or sets the option which indicates whether to enable intrabar order generation optimization with look-inside-bar back-testing.

```
BackTesting SetFillModel(tsopt.FillModel fillModel);
BackTesting SetFillVolume(int fillVolume);
BackTesting SetFillTrades(int fillTrades);
BackTesting SetMarketSlippage(double marketSlippage);
BackTesting SetPriceInsideBar(bool priceInsideBar);
BackTesting SetOptimizeIOG(bool optimizeIOG);
```

These methods perform the same operation as setting the equivalent property. However, they also return the `BackTesting` object so that you can chain the `Set…` methods together. This is useful when specifying options within a tree-style job definition.

OutSample Class

The `OutSample` class specifies the out-of-sample range (if any) for an optimization. It has the following members:

```
property tsopt.Settings EndOutSample { read; }
```

Returns the parent object in the job definition tree, which is a `Settings` object. This property is provided in order to support tree-style definitions.

```
property tsopt.SampleType SampleType { read; }
```

Gets the out-sample type. Note that this property is read-only. The sample type is set automatically when you set one of the other properties. The sample type property can have one of the following values:

- `tsopt.SampleType.ExcludeByPercent`
- `tsopt.SampleType.ExcludeByDate`

```
property int FirstPercent { read; write; }
```

Gets or sets the percentage of bars to exclude from the beginning of the history range.

If you *get* this property value and the sample type is not `ExcludeByPercent`, the property will throw an exception.

If you *set* this property value and the sample type is not `ExcludeByPercent`, the sample type will automatically be changed to `ExcludeByPercent` and the `LastPercent` will be set to zero.

`property int LastPercent { read; write; }`

Gets or sets the percentage of bars to exclude from the end of the history range.

If you *get* this property value and the sample type is not `ExcludeByPercent`, the property will throw an exception.

If you *set* this property value and the sample type is not `ExcludeByPercent`, the sample type will automatically be changed to `ExcludeByPercent` and the `FirstPercent` will be set to zero.

`property DateTime BeforeDate { read; write; }`

Gets or sets the first date to use for the in-sample range. Bars before this date will be part of the out-of-sample range.

If you *get* this property value and the sample type is not `ExcludeByDate`, the property will throw an exception.

If you *set* this property value and the sample type is not `ExcludeByDate`, the sample type will automatically be changed to `ExcludeByDate` and the `AfterDate` will be set to 12/31/9999 (so that no exclusion occurs at the end unless you set it explicitly).

`property DateTime AfterDate { read; write; }`

Gets or sets the last date to use for the in-sample range. Bars after this date will be part of the out-of-sample range.

If you *get* this property value and the sample type is not `ExcludeByDate`, the property will throw an exception.

If you *set* this property value and the sample type is not `ExcludeByDate`, the sample type will automatically be changed to `ExcludeByDate` and the `BeforeDate` will be set to the earliest possible date (so that no exclusion occurs at the beginning unless you set it explicitly).

```
OutSample SetFirstPercent(int percent);
OutSample SetLastPercent(int percent);
OutSample SetBeforeDate(DateTime date);
OutSample SetAfterDate(DateTime date);
```

These methods perform the same operation as setting the equivalent property. However, they also return the `OutSample` object so that you can chain the `Set…` methods together. This is useful when specifying options within a tree-style job definition.

`OutSample ExcludePercent(int firstPercent, int lastPercent);`

Sets the out-of-sample range by specifying the percentage of bars to exclude at the beginning and end. For example, to exclude the first 20 percent and the last 30 percent of bars from the in-sample, you would call `ExcludePercent(20, 30)`.

To use the entire range of bars as the in-sample, pass zero for both `firstPercent` and `lastPercent`. Note that this is the default setting.

```
// To use the entire range as the in-sample, don't exclude anything
job.Settings.OutSample.ExcludePercent(0, 0);
```

`OutSample ExcludeDates(DateTime beforeDate, DateTime afterDate);`

Sets the out-of-sample range by specifying the first and last dates of the in-sample. Bars prior to `beforeDate` or following `afterDate` will constitute the out-of-sample range.

WalkForward Class

The `WalkForward` class specifies the walk-forward options for an optimization. It has the following members:

`property tsopt.Settings EndWalkForward { read; }`

Returns the parent object in the job definition tree, which is a `Settings` object. This property is provided in order to support tree-style definitions.

`property bool Enabled { read; write; }`

Gets or sets the option which determines whether walk-forward optimization is enabled. Set this property to true if you wish to analyze the results of the optimization in the Walk-Forward Optimizer after the optimization has finished.

`property string TestName { read; write; }`

Gets or sets the test name for walk-forward optimization. This option is used only if walk-forward optimization is enabled.

`WalkForward SetEnabled(bool enabled);`
`WalkForward SetTestName(string testName);`

These methods perform the same operation as setting the equivalent property. However, they also return the `WalkForward` object so that you can chain the `Set…` methods together. This is useful when specifying options within a tree-style job definition.

CommissionBySymbol Class

The `CommissionBySymbol` class allows you to specify a different commission amount for each symbol in an optimization. This is useful when optimizing over symbols that have different commission rates. You can obtain a `CommissionBySymbol` object from the `CommissionBySymbol` property of a `CostsAndCapital` object.

The commissions for each symbol use the `CommissionMode` that you specify in the `CostsAndCapital` object. If you don't explicitly set the commission mode, the default is `tsopt.CommissionMode.FixedPerShare`.

If a symbol used in the optimization is not found in the `CommissionBySymbol` object, the optimizer will use the `CommissionAmount` specified in the `CostsAndCapital` object (which is zero by default).

The following code example shows how to define the commission per symbol when using the traditional coding style. Note that the specified commission rates are intended only as examples; they do not necessarily represent the current commission rates for your TradeStation account.

```
method void DefineCommission(tsopt.Job job)
vars:
   tsopt.CostsAndCapital costsAndCapital,
   tsopt.CommissionBySymbol commissionTable;
begin
   costsAndCapital = job.Settings.CostsAndCapital;

   costsAndCapital.CommissionMode = tsopt.CommissionMode.FixedPerShare;
   costsAndCapital.CommissionAmount = 0.01; // commission when symbol not found

   commissionTable = costsAndCapital.CommissionBySymbol;

   commissionTable["MSFT"] = 0.006;
   commissionTable["@ES"]  = 1.14;
   commissionTable["@BP"]  = 1.60;
end;
```

The next code example shows how to define the commission per symbol when using the tree style:

```
job.Settings
   .CostsAndCapital
      .SetCommissionMode(tsopt.CommissionMode.FixedPerShare)
      .SetCommissionAmount(0.01)
      .CommissionBySymbol
         .PutSymbolAmount("MSFT", 0.006)
         .PutSymbolAmount("@ES", 1.14)
         .PutSymbolAmount("@BP", 1.60)
      .EndCommissionBySymbol
   .EndCostsAndCapital
.UseDefinition();
```

You can also load the commission table from a text file. See the `ReadFile` method below for more information.

`property tsopt.CostsAndCapital EndCommissionBySymbol { read; }`

Returns the parent object in the job definition tree, which is a `CostsAndCapital` object. This property is provided in order to support tree-style definitions.

`property default double SymbolAmount[string] { read; write; }`

Gets or sets the commission for the specified symbol. The `CommissionBySymbol` class acts as a dictionary of amounts, and you can access the commission for a symbol using square brackets. For example, the following code sets the commission for MSFT:

```
job.Settings.CostsAndCapital.CommissionBySymbol["MSFT"] = 0.006;
```

The following code gets the commission for AAPL:

```
commission = job.Settings.CostsAndCapital.CommissionBySymbol["AAPL"];
```

When getting or setting the commission for multiple symbols, it's a good idea to save the `CommissionBySymbol` object in a local variable to avoid repetition. Then you can use square brackets directly on this variable:

```
method void DefineCommission(tsopt.Job job)
vars:
   tsopt.CommissionBySymbol commissionTable;
begin
   commissionTable = job.Settings.CostsAndCapital.CommissionBySymbol;

   commissionTable["MSFT"] = 0.006;
   commissionTable["AAPL"] = 0.01;
end;
```

`double GetSymbolAmount(string symbol);`

Gets the commission amount for the specified symbol. This is equivalent to accessing the commission with the square brackets syntax.

If the symbol is not found in the `CommissionBySymbol` object, this method will return zero.

`tsopt.CommissionBySymbol PutSymbolAmount(string symbol, double amount);`

Sets the commission amount for the specified symbol. This is equivalent to setting the commission with the square brackets syntax. However, this method also returns the `CommissionBySymbol` object, so you can chain together multiple `PutSymbolAmount` calls:

```
job.Settings
   .CostsAndCapital
      .CommissionBySymbol
         .PutSymbolAmount("MSFT", 0.006)
         .PutSymbolAmount("AAPL", 0.01);
```

`tsopt.CommissionBySymbol DeleteSymbolAmount(string symbol);`

Deletes the commission entry for the specified symbol. This method returns the `CommissionBySymbol` object, so you can chain together multiple `DeleteSymbolAmount` calls.

`tsopt.CommissionBySymbol Clear();`

Clears all the symbol entries from the `CommissionBySymbol` object.

`void ReadFile(string fileName);`

Reads the symbol entries from a text file. The `fileName` argument should specify the full path to the file.

Note that this method does *not* clear the existing symbol entries in the object. If the file contains a symbol that already has an entry in the object, the commission from the file will replace the existing commission for that symbol. If a symbol entry is not yet defined in the object, it will be added.

The text file should be formatted as a Comma-Separated-Values file with two columns. The first column should contain the symbol names, and the second column should contain the associated commission amounts. The file should not contain any column headings.

Here is an example of a properly formatted commission file:

```
MSFT,0.006
AAPL,0.01
@ES,1.14
@BP,1.6
```

The easiest way to create a commission file is to type the symbols and commissions into the first two columns of a spreadsheet and then save it as a CSV file. You can also create the file directly in a text editor. Finally, you can use the `WriteFile` method to write the contents of a `CommissionBySymbol` object that has been populated by your TradingApp.

`void WriteFile(string fileName);`

Writes the symbol entries in a `CommissionBySymbol` object to a text file. The `fileName` argument should specify the full path to the file.

This method writes the data using the format described under the `ReadFile` method above.

SlippageBySymbol Class

The `SlippageBySymbol` class allows you to specify a different slippage amount for each symbol in an optimization. This is useful when optimizing over symbols that typically have different slippage values. You can obtain a `SlippageBySymbol` object from the `SlippageBySymbol` property of a `CostsAndCapital` object.

The slippage values for each symbol use the `SlippageMode` that you specify in the `CostsAndCapital` object. If you don't explicitly set the slippage mode, the default is `tsopt.SlippageMode.FixedPerShare`.

If a symbol used in the optimization is not found in the `SlippageBySymbol` object, the optimizer will use the `SlippageAmount` specified in the `CostsAndCapital` object (which is zero by default).

The following code example shows how to define the slippage per symbol when using the traditional coding style. Note that the specified slippage rates are intended only as examples; they may not reflect the slippage amounts you would see in actual trading.

```
method void DefineSlippage(tsopt.Job job)
vars:
   tsopt.CostsAndCapital costsAndCapital,
   tsopt.SlippageBySymbol slippageTable;
begin
   costsAndCapital = job.Settings.CostsAndCapital;

   costsAndCapital.SlippageMode = tsopt.SlippageMode.FixedPerShare;
   costsAndCapital.SlippageAmount = 0.02; // slippage when symbol not found

   slippageTable = costsAndCapital.SlippageBySymbol;

   slippageTable["MSFT"] = 0.01;
   slippageTable["AAPL"] = 0.05;
   slippageTable["GOOG"] = 0.10;
end;
```

The next code example shows how to define the slippage per symbol when using the tree style:

```
job.Settings
   .CostsAndCapital
      .SetSlippageMode(tsopt.SlippageMode.FixedPerShare)
      .SetSlippageAmount(0.02)
      .SlippageBySymbol
         .PutSymbolAmount("MSFT", 0.01)
         .PutSymbolAmount("AAPL", 0.05)
         .PutSymbolAmount("GOOG", 0.10)
      .EndSlippageBySymbol
   .EndCostsAndCapital
.UseDefinition();
```

You can also load the slippage table from a text file. See the `ReadFile` method below for more information.

`property tsopt.CostsAndCapital EndSlippageBySymbol { read; }`

Returns the parent object in the job definition tree, which is a `CostsAndCapital` object. This property is provided in order to support tree-style definitions.

`property default double SymbolAmount[string] { read; write; }`

Gets or sets the slippage for the specified symbol. The `SlippageBySymbol` class acts as a dictionary of amounts, and you can access the slippage for a symbol using square brackets. For example, the following code sets the slippage for MSFT:

```
job.Settings.CostsAndCapital.SlippageBySymbol["MSFT"] = 0.01;
```

The following code gets the slippage for AAPL:

```
slippage = job.Settings.CostsAndCapital.SlippageBySymbol["AAPL"];
```

When getting or setting the slippage for multiple symbols, it's a good idea to save the `SlippageBySymbol` object in a local variable to avoid repetition. Then you can use square brackets directly on this variable:

```
method void DefineSlippage(tsopt.Job job)
vars:
   tsopt.SlippageBySymbol slippageTable;
begin
   slippageTable = job.Settings.CostsAndCapital.SlippageBySymbol;

   slippageTable["MSFT"] = 0.01;
   slippageTable["AAPL"] = 0.05;
end;
```

`double GetSymbolAmount(string symbol);`

Gets the slippage amount for the specified symbol. This is equivalent to accessing the slippage with the square brackets syntax.

If the symbol is not found in the `SlippageBySymbol` object, this method will return zero.

`tsopt.SlippageBySymbol PutSymbolAmount(string symbol, double amount);`

Sets the slippage amount for the specified symbol. This is equivalent to setting the slippage with the square brackets syntax. However, this method also returns the `SlippageBySymbol` object, so you can chain together multiple `PutSymbolAmount` calls:

```
job.Settings
   .CostsAndCapital
      .SlippageBySymbol
         .PutSymbolAmount("MSFT", 0.01)
         .PutSymbolAmount("AAPL", 0.05);
```

`tsopt.SlippageBySymbol DeleteSymbolAmount(string symbol);`

Deletes the slippage entry for the specified symbol. This method returns the `SlippageBySymbol` object, so you can chain together multiple `DeleteSymbolAmount` calls.

`tsopt.SlippageBySymbol Clear();`

Clears all the symbol entries from the `SlippageBySymbol` object.

`void ReadFile(string fileName);`

Reads the symbol entries from a text file. The `fileName` argument should specify the full path to the file.

Note that this method does *not* clear the existing symbol entries in the object. If the file contains a symbol that already has an entry in the object, the slippage from the file will replace the existing slippage for that symbol. If a symbol entry is not yet defined in the object, it will be added.

The text file should be formatted as a Comma-Separated-Values file with two columns. The first column should contain the symbol names, and the second column should contain the associated slippage amounts. The file should not contain any column headings.

Here is an example of a properly formatted slippage file:

```
MSFT,0.01
AAPL,0.05
GOOG,0.10
NFLX,0.04
```

The easiest way to create a slippage file is to type the symbols and slippage amounts into the first two columns of a spreadsheet and then save it as a CSV file. You can also create the file directly in a text editor. Finally, you can use the `WriteFile` method to write the contents of a `SlippageBySymbol` object that has been populated by your TradingApp.

`void WriteFile(string fileName);`

Writes the symbol entries in a `SlippageBySymbol` object to a text file. The `fileName` argument should specify the full path to the file.

This method writes the data using the format described under the `ReadFile` method above.

AvailableStrategies Helper Class

The `AvailableStrategies` class allows you to retrieve the names of all the strategies defined in the current TradeStation environment.

Note that this class is never used directly in a job definition. Instead, it is a helper class for applications that gather information from a user interface and use it to create a job definition. Suppose you want to allow the user to choose one or more strategies to optimize. In order to do that, you must get a list of all the available strategies so that you can present them to the user in a list box or combo box. The `AvailableStrategies` class provides a simple way to do that.

To use the `AvailableStrategies` class, just create an object of the class and iterate through it like a collection. You can use the `Count` property to get the number of items, and you can access each item by using array syntax on the object.

For example, the following code creates a `ComboBox` and populates it with all of the available strategies:

```
method ComboBox CreateStrategiesCombo(int x, int y, int width, int height)
vars:
   ComboBox combo,
   tsopt.AvailableStrategies availStrategies,
   int j;
begin
   combo = new ComboBox("", width, height);
   combo.Location(x, y);
   combo.DropDownStyle = ComboBoxStyle.dropdownlist;

   availStrategies = new tsopt.AvailableStrategies;
   for j = 0 to availStrategies.Count - 1 begin
      combo.AddItem(availStrategies[j]);
   end;

   return combo;
end;
```

The `AvailableStrategies` class has the following members:

`property int Count { read; }`

Returns the number of available strategies.

`property default string Strategy[int] { read; }`

Gets the strategy name at the specified zero-based index. The `AvailableStrategies` class acts as a collection of strategy names, and you can access the strategy name at a specific index using array syntax. For example, the following code gets the name of the first strategy in the current TradeStation environment:

```
availStrategies = new tsopt.AvailableStrategies;
name = availStrategies[0];
```

`method string GetStrategyName(int pos);`

Gets the strategy name at the specified zero-based index. This is an alternate way to retrieve the strategy name. For example, the following code shows another way to get the name of the first strategy:

```
availStrategies = new tsopt.AvailableStrategies;
name = availStrategies.GetStrategyName(0);
```

AvailableSessions Helper Class

The `AvailableSessions` class allows you to retrieve the names of all the custom market sessions defined in the current TradeStation environment.

Note that this class is never used directly in a job definition. Instead, it is a helper class for applications that gather information from a user interface and use it to create a job definition. Suppose you want to allow the user to choose a market session for the current optimization. In order to do that, you must get a list of all the available sessions so that you can present them to the user in a list box or combo box. The `AvailableSessions` class provides a simple way to do that.

Note: The default session is represented by an empty string. (This is the default value of the `SecurityOptions.SessionName` property.) Thus, the `AvailableSessions` class does not return the name of the default session, since it is always an empty string.

To use the `AvailableSessions` class, just create an object of the class and iterate through it like a collection. You can use the `Count` property to get the number of items, and you can access each item by using array syntax on the object.

For example, the following code creates a `ComboBox` and populates it with all of the available market sessions:

```
method ComboBox CreateSessionsCombo(int x, int y, int width, int height)
vars:
   ComboBox combo,
   tsopt.AvailableSessions availSessions,
   int j;
begin
   combo = new ComboBox("", width, height);
   combo.Location(x, y);
   combo.DropDownStyle = ComboBoxStyle.dropdownlist;

   // Add a string to represent the default session
   combo.AddItem("Default Session");

   availSessions = new tsopt.AvailableSessions;
   for j = 0 to availSessions.Count - 1 begin
      combo.AddItem(availSessions[j]);
   end;

   return combo;
end;
```

Note: If you want to include the default session in the combo box, your code must add it explicitly, as shown in the example above; it is not returned by the `AvailableSessions` class. If the user chooses the "Default Session" option (or whatever you decide to call it), your code should assign an empty string to the `SecurityOptions.SessionName` property instead. This substitution is required for the default session, since it is always represented by an empty string in the Optimization API.

The `AvailableSessions` class has the following members:

`property int Count { read; }`

Returns the number of available sessions.

`property default string Session[int] { read; }`

Gets the session name at the specified zero-based index. The `AvailableSessions` class acts as a collection of session names, and you can access the session name at a specific index using array syntax. For example, the following code gets the name of the first session in the current TradeStation environment:

```
availSessions = new tsopt.AvailableSessions;
name = availSessions[0];
```

`method string GetSessionName(int pos);`

Gets the session name at the specified zero-based index. This is an alternate way to retrieve the session name. For example, the following code shows another way to get the name of the first session:

```
availSessions = new tsopt.AvailableSessions;
name = availSessions.GetSessionName(0);
```

Optimization Classes

The following classes are used to manage an optimization and to provide information about its progress.

Optimizer Class

The `Optimizer` class provides methods to start or cancel an optimization. This is also where you define the event handlers for the optimization.

`static tsopt.Optimizer Create();`

Creates a new `Optimizer` object:

```
optimizer = tsopt.Optimizer.Create();
```

You may also use the `new` keyword to create the object:

```
optimizer = new tsopt.Optimizer;
```

`JobDone Event`

The optimizer calls your `JobDone` event handler when the job is finished, either because it completed normally or because it was canceled. Note that if an error occurred during the optimization, the `JobFailed` event handler is called instead.

Before starting an optimization, you must assign an event handler to the `JobDone` event. The event handler should be defined as follows. (You may use whatever method name you wish.)

```
method void OptDone(object sender, tsopt.JobDoneEventArgs args)
begin
   // Handle event here
end;
```

After creating the `Optimizer` object, be sure to assign the event handler to the `JobDone` event:

```
optimizer.JobDone += OptDone;
```

The `JobDone` event handler is where you retrieve and process the results of an optimization. The results are available as a property of the `JobDoneEventArgs` argument.

If your event handler calls many methods and/or properties on the `Results` object, it's a good idea to save the object in a local variable. This will make your code both more efficient and more concise:

```
method void OptDone(object sender, tsopt.JobDoneEventArgs args)
vars:
   tsopt.Results results;
begin
   results = args.Results;

   // Call methods or properties of the results variable
end;
```

You can determine whether an optimization job was completed or canceled by checking the `Canceled` property of the `JobDoneEventArgs` argument. If the job was canceled, the `Results` object will contain any results that were processed before the cancellation.

JobFailed Event

The optimizer calls your `JobFailed` event handler when an error occurs during an optimization.

Before starting an optimization, you must assign an event handler to the `JobFailed` event. The event handler should be defined as follows. (You may use whatever method name you wish.)

```
method void OptError(object sender, tsopt.JobFailedEventArgs args)
begin
   // Handle event here
end;
```

After creating the `Optimizer` object, be sure to assign the event handler to the `JobFailed` event:

```
optimizer.JobFailed += OptError;
```

A `JobFailed` event handler should report the optimization error to the user. The error object can be retrieved from the `Error` property of the `JobFailedEventArgs` argument. You can get the error message by accessing the `Message` property of the error object:

```
method void OptError(object sender, tsopt.JobFailedEventArgs args)
begin
   txtStatus.Text = "Optimization Error: " + args.Error.Message;
end;
```

ProgressChanged Event

The optimizer calls your `ProgressChanged` event handler periodically during an optimization in order to provide information about the progress of the optimization.

The `ProgressChanged` event handler is optional, but it is a good idea to provide it. The event handler should be defined as follows. (You may use whatever method name you wish.)

```
method void OptProgress(object sender, tsopt.ProgressChangedEventArgs args)
begin
   // Handle event here
end;
```

After defining the event handler, be sure to add it to the optimizer object:

```
optimizer.ProgressChanged += OptProgress;
```

You can retrieve information about the current state of the optimization by accessing the `Progress` and `BestValues` properties of the `ProgressChangedEventArgs` argument. For more information about the objects returned by these properties, see the entries below for the `ProgressInfo` and `BestValues` classes.

The following `ProgressChanged` event handler outputs information about the current test number and best fitness to the TradeStation Print Log:

```
method void OptProgress(Object sender, tsopt.ProgressChangedEventArgs args)
begin
   Print("Test ", args.Progress.TestNum.ToString(), " of ",
         args.Progress.TestCount.ToString());
   Print("   ", args.BestValues.FitnessName, " = ",
         args.BestValues.FitnessValue.ToString());
end;
```

`int64 StartJob(tsopt.Job job);`

Starts an optimization using the specified job definition. The optimization runs asynchronously in the background and notifies your application about its status by calling your event handlers. Since the optimization is asynchronous, the `StartJob` method returns immediately.

The value returned by the method is a 64-bit integer that identifies the job. If you are queuing multiple optimizations, you can use this integer to identify the job. (For example, you can pass it to the `CancelJob` method to cancel that particular job.) If you are not queuing optimizations, you can simply ignore the return value; you don't need to assign it to a variable.

This version of the `StartJob` method automatically determines how many threads to use for the optimization, based on the number of processor cores in your machine.

Before it begins the optimization, the `StartJob` method will validate the job definition to ensure that it is complete and correct. If there are any errors in the job definition, the method will throw a `tsopt.OptimizationException`. If you do not catch the exception, the application will halt and the error will be displayed in the TradeStation Events Log. However, you may also handle the error yourself by putting the `StartJob` call inside a `try...catch` block:

```
try
   optimizer.StartJob(job);
catch (tsopt.OptimizationException error)
   // Retrieve info from error object and handle error
end;
```

`int64 StartJob(tsopt.Job job, int numThreads);`

Starts an optimization using the specified job definition and the specified number of threads. Note that the optimizer will not use more threads than the number of processor cores, since there is no advantage to doing this. Thus, you can use this method to *decrease* the number of threads assigned to an optimization, but you cannot force the optimizer to use more threads than the single-argument version of the `StartJob` method. If the value of `numThreads` is higher than the number of cores, the optimizer will simply use the number of cores instead.

`void CancelJob();`

Cancels the optimization that is currently running. When the optimizer has canceled the optimization, it will call your `JobDone` event handler. The results for all completed optimization tests will be available in the `tsopt.Results` object.

`void CancelJob(int64 jobID);`

Cancels the optimization with the specified `jobID` (which is returned by the `StartJob` method). This version of the `CancelJob` method allows you to cancel a specific job if you have queued multiple optimizations. If the specified job is currently running, the optimizer will cancel it and call your `JobDone` event handler. If the job is waiting in the queue, the optimizer will simply remove it from the queue.

`void CancelAllJobs();`

Cancels all optimization jobs for the optimizer, including the currently running job and any jobs that are waiting in the queue.

This method provides a convenient way to cancel all jobs when you have queued multiple optimizations. The optimizer will stop the currently running optimization and call your `JobDone` event handler. Any waiting jobs will simply be removed from the queue.

JobDoneEventArgs Class

A `JobDoneEventArgs` object is passed as the second argument of your `JobDone` event handler. It provides information about the optimization job that just finished.

`property int64 JobID { read; }`

Returns a 64-bit integer that identifies the optimization job that just finished. This will be the same value that was returned from the `StartJob` call that started the optimization.

The `JobID` property allows you to identify which job has completed when you have queued multiple optimizations. If you are not queuing optimizations, you can ignore it.

`property tsopt.Results Results { read; }`

Returns a `tsopt.Results` object that holds the results for the optimization job that just finished. You can call methods and properties on this object to retrieve the results.

For a detailed discussion about working with results, see *Retrieving Optimization Results* earlier in this guide.

For complete information about the `Results` class, see its entry below in the *API Reference*.

`property bool Canceled { read; }`

Returns true if the `JobDone` event handler was called because the optimization job was canceled.

Note that when an optimization is canceled, the `Results` object will contain any results that were processed before the cancellation.

JobFailedEventArgs Class

A `JobFailedEventArgs` object is passed as the second argument of your `JobFailed` event handler. It provides information about the optimization job that just failed.

JobFailedEventArgs Class (continued)

property int64 JobID { read; }

Returns a 64-bit integer that identifies the optimization job that just failed. This will be the same value that was returned from the `StartJob` call that started the optimization.

The `JobID` property allows you to identify which job has failed when you have queued multiple optimizations. If you are not queuing optimizations, you can ignore it.

property tsopt.OptimizationException Error { read; }

Returns a `tsopt.OptimizationException` object that provides error information about the job that just failed. You can access the `Message` property of this object to get the error message.

ProgressChangedEventArgs Class

A `ProgressChangedEventArgs` object is passed as the second argument of your `ProgressChanged` event handler. It provides information about the progress of the optimization that is currently running.

property int64 JobID { read; }

Returns a 64-bit integer that identifies the optimization job that is currently running. This will be the same value that was returned from the `StartJob` call that started the optimization.

The `JobID` property allows you to identify which job is running when you have queued multiple optimizations. If you are not queuing optimizations, you can ignore it.

property tsopt.ProgressInfo Progress { read; }

Returns a `tsopt.ProgressInfo` object that contains information about the progress of the running optimization, such as the current test number.

For complete information about the `ProgressInfo` class, see its entry below in the *API Reference*.

property tsopt.BestValues BestValues { read; }

Returns a `tsopt.BestValues` object that contains information about the best fitness result seen so far in the optimization.

For complete information about the `BestValues` class, see its entry below in the API.

ProgressInfo Class

The `ProgressInfo` class provides progress information about the optimization that is current running.

You can get a `ProgressInfo` object by accessing the `Progress` property of the `ProgressChangedEventArgs` object, which is passed as the second argument of your `ProgressChanged` event handler.

property double ElapsedTime { read; }

Returns the time in seconds that the current optimization has been running.

property double RemainingTime { read; }

Returns the estimated time in seconds that will be required for the current optimization to finish.

property int TestNum { read; }

Returns the highest test number that has been evaluated so far in the current optimization.

property int TestCount { read; }

Returns the total number of tests that will be evaluated in the current optimization.

property bool WaitingForData { read; }

Returns true if the optimization is currently waiting for data to be downloaded from the TradeStation network. If the entire history range specified by the job definition is not in your local cache, the optimizer will request that the missing data be downloaded ASAP before starting the optimization. It's a good idea for your application to check the `WaitingForData` property and display an appropriate status message when this is happening; otherwise, it might appear that the optimization has stalled. Note that when the optimizer requests data from the network, you can view the pending data requests in the Download Scheduler window, just as you can for Charting. However, the optimizer will always request the data ASAP (even if your default setting is "Off Peak"), since it needs the missing data in order to perform the optimization on the requested history range.

property int GenerationNum { read; }

For a genetic optimization, returns the current generation that is being optimized. (This property always returns zero for an exhaustive optimization.)

property int GenerationCount { read; }

For a genetic optimization, returns the total number of generations that will be optimized. (This property always returns zero for an exhaustive optimization.)

property int IndivNum { read; }

For a genetic optimization, returns the number of the highest individual evaluated so far in the current generation. (This property always returns zero for an exhaustive optimization.)

property int PopSize { read; }

For a genetic optimization, returns the size of the population. (This property always returns zero for an exhaustive optimization.)

BestValues Class

The `BestValues` class contains information about the best fitness result seen so far, as well as the optimized parameters that produced that fitness result.

You can get a `BestValues` object by accessing the `BestValues` property of the `ProgressChangedEventArgs` object, which is passed as the second argument of your `ProgressChanged` event handler.

`property string FitnessName { read; }`

Returns the name of the fitness function that is being used for this optimization.

`property double FitnessValue { read; }`

Returns the best fitness value seen so far in this optimization.

`property int OptParamCount { read; }`

Returns the number of optimized parameters for this optimization. Optimized parameters are often inputs, but they can also be symbols, intervals, or the enabled status of a strategy.

`method string GetOptParamName(int param);`

Gets the name of the specified optimized parameter. The `param` argument is the zero-based index of the parameter. You can loop from `0` to `OptParamCount-1` to get all the optimized parameter names.

For an input, the name will include the strategy name and input name, e.g. "Bollinger Bands LE: Length".

For a symbol, the name will include the data series, e.g. "Data1: Symbol".

For an interval, the name will include the data series, e.g. "Data1: Interval".

For a strategy enabled status, the name will be "Enable" followed by the strategy name, e.g. "Enable Bollinger Bands LE".

`method string GetOptParamValue(int param);`

Gets the value of the specified optimized parameter as a string. The `param` argument is the zero-based index of the parameter. You can loop from `0` to `OptParamCount-1` to get all the optimized parameter values that produced the best fitness result so far.

The value is returned as a string because the optimized parameter may be either numeric or non-numeric. If the value is numeric, the method returns the string representation of that number.

For an input, the method will return the input value, e.g. "25" or "Close".

For a symbol, the method will return the symbol name, e.g. "MSFT".

For an interval, the method will return a description of the interval, e.g. "5 min".

For a strategy enabled status, the method will return either "true" or "false".

OptimizationException Class

The Optimization API reports most errors via the `OptimizationException` class:

- If there is a non-recoverable error in your job definition code, the API will throw an `OptimizationException` from that code.
- If there is a validation error in your job definition, the API will throw an `OptimizationException` from the `Optimizer.StartJob` method.
- If an error occurs during an optimization, the optimizer will call your `JobFailed` event handler. You can retrieve the error from the `args.Error` property, which returns an `OptimizationException` object.

The `OptimizationException` class inherits from the `elsystem.Exception` class, so it offers all the same properties and methods as that class. In addition, it provides some properties of its own that you may find useful.

Here are the key properties of this class:

`property string Message { read; }`

This property is inherited from the `Exception` class. It returns the full error message for the exception. For job validation errors, this message will include the "path" down the job tree to the node that caused the error. This can be very helpful for debugging.

`property string MessageNoPath { read; }`

This is a special property provided by the `OptimizationException` class. It returns the error message, but it omits the "error path" for job validation errors. You may wish to use this property if you want to report validation errors to an end user. Most validation errors result from problems in a job definition, and these can be corrected by fixing the code. However, some validation errors can be caused by a problem in the environment; for example, a strategy required by the optimization may be missing from the current TradeStation environment.

Note that errors that occur *during* an optimization never include the "error path," since these errors are rarely caused by problems in the job definition. Thus, the `Message` and `MessageNoPath` properties will always return the same message within a `JobFailed` event handler.

`property tsopt.ErrorCode ErrorCode { read; }`

This is a special property provided by the `OptimizationException` class. It returns a value from the `tsopt.ErrorCode` enumeration that specifies what kind of error occurred. This can be useful if you want to check for specific errors in your code and handle them in a special way.

For example, the following code catches any job validation error thrown by the `StartJob` method. If the validation error is an invalid strategy name, then the exception handler displays the error message in a text box. Otherwise, it re-throws the exception so that the application will halt and show the error in the TradeStation Events Log.

Notice that the code uses the `MessageNoPath` property so that the user sees the error message without the "error path."

```
try
   optimizer.StartJob(job);
catch (tsopt.OptimizationException error)
   if error.ErrorCode = tsopt.ErrorCode.InvalidStrategyName then
      txtStatus.Text = error.MessageNoPath // display the error
   else
      throw error; // re-throw the exception
end;
```

To see the available values for the `ErrorCode` enumeration, search for "ErrorCode" in the Dictionary window of the TradeStation Development Environment; then click on the `tsopt.ErrorCode` entry.

The Results Class

The `tsopt.Results` class provides complete information about the results of an optimization.

When the optimizer calls your `JobDone` event handler, it passes a `JobDoneEventArgs` object as the second argument (`args`). You can access the `Results` property of this object to get the `Results` object for the optimization.

For a detailed discussion about using the `Results` class, see *Retrieving Optimization Results* earlier in this guide.

`property int TestCount { read; }`

Returns the number of optimization tests in the `Results` object.

Note that you can specify the number of tests to keep by setting the `NumTestsToKeep` property in the `ResultOptions` object, which is a sub-object of `Settings`. (The default value of `NumTestsToKeep` is 200.) However, the value returned by `TestCount` will not necessarily be the same as this number. It may be *less* if the number of parameter combinations is lower than `NumTestsToKeep`. The count can also be *higher* if there are multiple tests with identical fitness at the bottom of the results. (The `Results` object will include all of the "fitness ties" at the bottom, even if this causes the number of tests to be higher than requested. However, the number of tests will never exceed twice the requested number.) Thus, you should always use the `TestCount` property to determine the actual number of tests in the `Results` object.

`int GetTestNum(int index);`

Returns the test number for the test at the specified index in the results.

Note that the test *index* (which is passed as an argument) specifies the position of the test in the `Results` object: 0 indicates the first test, 1 indicates the second test, and so on. This number must always be less than `TestCount`.

On the other hand, the test *number* (which is returned by this method) identifies where the test appeared in the sequence of all tests that were evaluated by the optimizer. This number can be much higher than `TestCount`, since it identifies the test among *all* the evaluated tests, not just the top tests that were retained in the final results.

`string GetTestSymbol(int index);`
`string GetTestSymbol(int index, int dataNum);`

Returns the symbol that was used for the test at the specified index in the results.

The first version of this method returns the symbol used for the primary data series (Data1).

For multi-data strategies, you can use the second version of this method, which accepts the data series as a second argument (1 for Data1, 2 for Data2, etc.). For example, the following code returns the symbol used for the first test for Data2:

```
sym = results.GetTestSymbol(0, 2);
```

```
tsopt.Interval GetTestInterval(int index);
tsopt.Interval GetTestInterval(int index, int dataNum);
```

Returns the interval that was used for the test at the specified index in the results.

The first version of this method returns the interval used for the primary data series (Data1).

For multi-data strategies, you can use the second version of this method, which accepts the data series as a second argument (1 for Data1, 2 for Data2, etc.). For example, the following code returns the interval used for the first test for Data2:

```
interval = results.GetTestInterval(0, 2);
```

The interval is returned as a `tsopt.Interval` object. If you want a string representation of the interval, you can call the `Describe` method on the object. You may also call other methods or properties on the `Interval` object to get detailed information about the interval.

```
string GetTestInputString(int index, string inputName,
   string strategyName);
string GetTestInputString(int index, string inputName,
   string strategyName, int strategyInstance);
```

Returns the value of an input to the specified strategy for the test at the specified index in the results.

The `strategyName` argument must be the name of one of the strategies defined in the optimization job. Otherwise, the method will throw an `OptimizationException` with a message that the strategy was not found.

The `inputName` argument must be the name of one of the inputs in that strategy. Otherwise, the method will throw an `OptimizationException` with a message that the input was not found.

If the optimization job defines more than one instance of the specified strategy, you can use the second version of this method and specify the `strategyInstance` to use (0 for the first instance, 1 for the second instance, and so on). The first version of this method always uses the first instance of the specified strategy.

The `GetTestInputString` method returns the input value as a string, so it is flexible enough to handle any kind of input, including text inputs, expression inputs, and Boolean inputs.

```
double GetTestInputDouble(int index, string inputName,
   string strategyName);
double GetTestInputDouble(int index, string inputName,
   string strategyName, int strategyInstance);
```

Returns the numeric value of an input to the specified strategy for the test at the specified index in the results.

This method works just like `GetTestInputString`, but it returns the input value as a double, saving you the trouble of converting a string to a number. You should use this method only if you

know that the specified input is numeric; otherwise, the method will throw an exception to indicate that it cannot convert the input value to a double.

```
bool GetTestInputBool(int index, string inputName,
   string strategyName);
bool GetTestInputBool(int index, string inputName,
   string strategyName, int strategyInstance);
```

Returns the Boolean value of an input to the specified strategy for the test at the specified index in the results.

This method works just like `GetTestInputString`, but it returns the input value as a Boolean, saving you the trouble of converting a string to a True/False value. You should use this method only if you know that the specified input is Boolean; otherwise, the method will throw an exception to indicate that it cannot convert the input value to a Boolean.

```
bool IsTestStrategyEnabled(int index, string strategyName);
bool IsTestStrategyEnabled(int index, string strategyName,
   int strategyInstance);
```

Returns true if the specified strategy was enabled for the test at the specified index.

This method can be helpful if you used the `OptStrategyEnabled` method to optimize the enabled status of a strategy, and you want to know whether the strategy was enabled or disabled for a particular test.

If the optimization job defines more than one instance of the specified strategy, you can use the second version of this method and specify the `strategyInstance` to use (0 for the first instance, 1 for the second instance, and so on). The first version of this method always uses the first instance of the specified strategy.

```
DateTime GetSampleStartDate();
DateTime GetSampleStartDate(int index);
```

Returns the start date and time of the in-sample range used for the optimization.

This method can be helpful if you used percentages to specify the out-of-sample range, and you want to know exactly when the resulting in-sample range started.

If you are *not* optimizing over symbols, the first version of this method will give you the in-sample start date for the entire optimization.

However, if you *are* optimizing over symbols, you will need to use the second version of this method to get the in-sample start date for each test. This is necessary because different symbols can have different bar counts for the same history range, and this can produce different in-sample ranges when the out-of-sample range is defined by percentages.

```
DateTime GetSampleEndDate();
DateTime GetSampleEndDate(int index);
```

Returns the end date and time of the in-sample range used for the optimization.

This method can be helpful if you used percentages to specify the out-of-sample range, and you want to know exactly when the resulting in-sample range ended.

If you are *not* optimizing over symbols, the first version of this method will give you the in-sample end date for the entire optimization.

However, if you *are* optimizing over symbols, you will need to use the second version of this method to get the in-sample end date for each test. This is necessary because different symbols can have different bar counts for the same history range, and this can produce different in-sample ranges when the out-of-sample range is defined by percentages.

```
property int OptParamCount { read; }
```

Returns the number of optimized parameters.

Note that optimized parameters are not necessarily inputs; they can also be symbols, intervals, or the enabled status of a strategy. This method returns the count of *all* optimized parameters for the current job, not just optimized inputs.

```
string GetOptParamHeading(int param);
```

Returns a descriptive heading for the specified optimized parameter.

The `param` argument is the zero-based index of the optimized parameter. It should always be less than `OptParamCount`.

For an optimized input, the heading will include the strategy name and input name, e.g. “Bollinger Bands LE: Length”.

For an optimized symbol, the heading will include the data series and the word “Symbol”, e.g. “Data1: Symbol”.

For an optimized interval, the heading will include the data series and the word “Interval”, e.g. “Data1: Interval”.

For an optimized strategy-enabled status, the heading will be the word “Enable” followed by the strategy name, e.g. “Enable Stop Loss”.

```
string GetOptValueString(int param, int index);
```

Returns a string representation of the value for a particular optimized parameter and test.

The `param` argument is the zero-based index of the optimized parameter. It should always be less than `OptParamCount`.

The index argument is the zero-based index of the test within the results. It should always be less than `TestCount`.

For an optimized input, this method will return the value as a string. For example, a numeric value such as 15 will be returned as the string “15”. A text value will also be returned as a string, but the

value will be surrounded by embedded double-quotes, e.g. “"ABCDE"” An input expression such as AvgPrice will be returned as “AvgPrice” (without embedded quotes).

For an optimized symbol, this method will simply return the symbol, e.g. “MSFT”.

For an optimized interval, this method will return a description of the interval, e.g. “5 min” or “Daily”.

For an optimized enabled status, this method will return either “true” or “false”.

`static string GetMetricName(tsopt.MetricID metricID);`

Returns the metric name for the specified `MetricID`.

This is a static method, so you should call it as `tsopt.Results.GetMetricName`.

For example, the following code will assign “Net Profit” to the `name` variable:

```
name = tsopt.Results.GetMetricName(tsopt.MetricID.NetProfit);
```

`static string GetMetricHeading(tsopt.MetricID metricID,`
`tsopt.TradeType type);`

Returns the metric heading for the specified `MetricID` and `TradeType`. The heading will prefix the name of the metric with the trade type and a colon, like the metric headings in the Strategy Optimization Report.

This is a static method, so you should call it as `tsopt.Results.GetMetricHeading`.

For example, the following code will assign “Long: Net Profit” to the `heading` variable:

```
heading = tsopt.Results.GetMetricHeading(tsopt.MetricID.NetProfit,
   tsopt.TradeType.LongTrades);
```

`double GetMetric(tsopt.MetricID metricID, int index,`
`tsopt.TradeType type, tsopt.ResultsRange range);`

Returns the value of the specified `MetricID` for the specified test index, trade type, and results range.

For example, the following code will get the Gross Profit for the fifth test for short trades and out-of-sample results:

```
grossProfit = results.GetMetric(tsopt.MetricID.GrossProfit, 4,
   tsopt.TradeType.ShortTrades, tsopt.ResultsRange.OutSample);
```

The `GetMetric` method is the most general and flexible way to retrieve a metric value from the results, since it allows you to specified the desired metric as an argument. For example, you can loop through all the MetricIDs and retrieve their values with the `GetMetric` method. For an example of this technique, see *Example: Writing the Results to a Text File* earlier in this guide.

You can also retrieve results using named metric methods, which can be more concise and readable if your code needs to access a specific metric. The named metric methods are described later in this section.

`void SortByMetric(tsopt.MetricID metricID, tsopt.TradeType type, tsopt.ResultsRange range, bool reverse);`

Sorts the optimization tests by the specified metric, using the values for the specified trade type and results range. The `reverse` argument indicates whether to sort the tests in reverse order (highest to lowest).

For example, the following code sorts the results by Profit Factor in descending order (from largest to smallest values). It uses the in-sample results for all trades (long and short):

```
results.SortByMetric(tsopt.MetricID.ProfitFactor, tsopt.TradeType.AllTrades,
   tsopt.ResultsRange.InSample, true {reverse});
```

`void SortByTestNum(bool reverse);`

Sorts the optimization tests by test number. The `reverse` argument indicates whether to sort the tests in reverse order (highest to lowest).

The test number is the value returned by the `GetTestNum` method. It identifies where the test appeared in the sequence of all tests that were evaluated by the optimizer.

For example, the following code sorts the results by test number in ascending order:

```
results.SortByTestNum(false);
```

`void ApplyTestToJob(tsopt.Job job, int index);`

Applies the values of the optimized parameters from the specified test to the specified job definition. This replaces the optimized parameters in the job definition with the fixed values from the test.

The `job` argument should be the job definition that produced the optimization results. This is important because `ApplyTestToJob` expects the optimized values in the test results to correspond to optimized parameters in the job definition. If this is not the case, you will probably get an error, either when you call `ApplyTestToJob` or when you try to run the modified job.

The `index` argument should specify the zero-based index of the test you wish to apply to the job.

For example, suppose a job definition optimizes “Bollinger Bands LE” by the “NumDevsDn” input and “Bollinger Bands SE” by the “NumDevsUp” input:

```
strategy = job.Strategies.AddStrategy("Bollinger Bands LE");
strategy.ELInputs.OptRange("NumDevsDn", 1, 3, 0.25);

strategy = job.Strategies.AddStrategy("Bollinger Bands SE");
strategy.ELInputs.OptRange("NumDevsUp", 1, 3, 0.25);
```

Now suppose that we optimize this job and call `ApplyTestToJob` in our `JobDone` event handler:

```
results.ApplyTestToJob(job, 0);
```

Since we specified the first test in the results (i.e., index = 0), this call will use the test with the best fitness. The `ApplyTestToJob` method will take this test's values for "NumDevsDn" and "NumDevsUp" and change the job definition to use fixed values for these inputs. If the input values for the first test are 1.5 and 2.5 respectively, then the `ApplyTestToJob` call above is equivalent to the following code:

```
job.Strategies[0].ELInputs.SetInput("NumDevsDn", 1.5);
job.Strategies[1].ELInputs.SetInput("NumDevsUp", 2.5);
```

In other words, `ApplyTestToJob` replaces the optimized inputs (which were specified by `OptRange`) with the fixed input values from the test.

The Optimization API allows you to optimize other kinds of parameters besides inputs, and `ApplyTestToJob` works equally well for these. For example, if a job optimizes over symbols, then the `ApplyTestToJob` method will replace the optimized symbol in the job definition with the fixed symbol from the specified test in the results.

The `ApplyTestToJob` method is typically used when you want to use the results of one optimization as the starting point for another optimization. For example, suppose you want to take the best result from the optimization above and then perform another optimization over various stop loss amounts. You could define the new job as follows:

```
results.ApplyTestToJob(job, 0);
strategy = job.Strategies.AddStrategy("Stop Loss");
strategy.ELInputs.OptRange("Amount", 1, 5, 1);
```

If you optimize this revised job definition, it will use fixed inputs for "NumDevsDn" and "NumDevsUp" and optimize over the "Amount" input in the "Stop Loss" strategy instead.

`void WriteFile(string fileName, string delimiter, tsopt.TradeType type, tsopt.ResultsRange range);`

Writes the optimization results for the specified trade type and results range to a text file. The file will have the same basic format as the Strategy Optimization Report in TradeStation Charting.

The `fileName` argument should specify the full path to the file you want to write.

The `delimiter` argument should specify the character (or sequence of characters) that separates each value on a line. If you pass a comma as the delimiter, the `WriteFile` method will write a valid Comma-Separated-Values file: it will surround values with quotes if necessary (e.g., when a value contains a comma), and it will escape any embedded quotes in a value. This allows the file to be opened with any program that knows how to read CSV files, such as Microsoft Excel®.

The `type` and `range` arguments should specify the trade type and the results range to use when writing the results.

Named Metric Methods

Each of the methods listed below returns a specific named metric from the optimization results.

There are two versions of each named metric method:

1. The first version takes no arguments and returns the metric value for the first test in the results, for all trades, and for the entire history range. Note that the results are initially sorted in order of best to worst fitness. Thus, the first test will be the one with the best fitness value (unless you sort the results into a different order).
2. The second version returns the metric value for the specified test index, trade type, and results range.

Calling a named metric function is equivalent to calling the `GetMetric` function and passing the relevant `MetricID` value For example, the following three calls will return the same result:

```
profit = results.NetProfit();

profit = results.NetProfit(0, tsopt.TradeType.AllTrades,
   tsopt.ResultsRange.All);

profit = results.GetMetric(tsopt.MetricID.NetProfit, 0,
   tsopt.TradeType.AllTrades, tsopt.ResultsRange.All);
```

The `GetMetric` method is more general and flexible than the named metric methods. However, the named metrics can be more convenient and readable when you want to retrieve specific metrics in your code.

Here is a list of all the named metric methods in the `tsopt.Results` class:

```
double NetProfit();
double NetProfit(int index, tsopt.TradeType type, tsopt.ResultsRange range);

double GrossProfit();
double GrossProfit(int index, tsopt.TradeType type, tsopt.ResultsRange range);

double GrossLoss();
double GrossLoss(int index, tsopt.TradeType type, tsopt.ResultsRange range);

int TotalTrades();
int TotalTrades(int index, tsopt.TradeType type, tsopt.ResultsRange range);

double PercentProfitable();
double PercentProfitable(int index, tsopt.TradeType type,
   tsopt.ResultsRange range);

int WinningTrades();
int WinningTrades(int index, tsopt.TradeType type, tsopt.ResultsRange range);

int LosingTrades();
int LosingTrades(int index, tsopt.TradeType type, tsopt.ResultsRange range);

double MaxWinningTrade();
double MaxWinningTrade(int index, tsopt.TradeType type,
   tsopt.ResultsRange range);
```

```
double MaxLosingTrade();
double MaxLosingTrade(int index, tsopt.TradeType type,
   tsopt.ResultsRange range);

double AvgWinningTrade();
double AvgWinningTrade(int index, tsopt.TradeType type,
   tsopt.ResultsRange range);

double AvgLosingTrade();
double AvgLosingTrade(int index, tsopt.TradeType type,
   tsopt.ResultsRange range);

double WinLossRatio();
double WinLossRatio(int index, tsopt.TradeType type, tsopt.ResultsRange range);

double AvgTrade();
double AvgTrade(int index, tsopt.TradeType type, tsopt.ResultsRange range);

int MaxConsecutiveWinners();
int MaxConsecutiveWinners(int index, tsopt.TradeType type,
   tsopt.ResultsRange range);

int MaxConsecutiveLosers();
int MaxConsecutiveLosers(int index, tsopt.TradeType type,
   tsopt.ResultsRange range);

int AvgBarsInWinner();
int AvgBarsInWinner(int index, tsopt.TradeType type,
   tsopt.ResultsRange range);

int AvgBarsInLoser();
int AvgBarsInLoser(int index, tsopt.TradeType type,
   tsopt.ResultsRange range);

double MaxIntradayDrawdown();
double MaxIntradayDrawdown(int index, tsopt.TradeType type,
   tsopt.ResultsRange range);

double ProfitFactor();
double ProfitFactor(int index, tsopt.TradeType type, tsopt.ResultsRange range);

double MaxContractsHeld();
double MaxContractsHeld(int index, tsopt.TradeType type,
   tsopt.ResultsRange range);

double RequiredAccountSize();
double RequiredAccountSize(int index, tsopt.TradeType type,
   tsopt.ResultsRange range);

double ReturnOnAccount();
double ReturnOnAccount(int index, tsopt.TradeType type,
   tsopt.ResultsRange range);

double TSIndex();
double TSIndex(int index, tsopt.TradeType type, tsopt.ResultsRange range);

double ExpectancyScore();
double ExpectancyScore(int index, tsopt.TradeType type,
   tsopt.ResultsRange range);
```

```
double PessimisticReturnOnCapital();
double PessimisticReturnOnCapital(int index, tsopt.TradeType type,
   tsopt.ResultsRange range);

double PerfectProfitCorrelation();
double PerfectProfitCorrelation(int index, tsopt.TradeType type,
   tsopt.ResultsRange range);

double RobustnessIndex();
double RobustnessIndex(int index, tsopt.TradeType type,
   tsopt.ResultsRange range);
```

Appendix 1: Anatomy of a Tree-Style Definition

The "tree style" was introduced in this guide as an optional way to define an optimization job. Although it's possible to study some examples and get an intuitive sense of how to write a tree-style definition, it can be helpful to understand how the tree style works "under the hood." This appendix analyzes a typical definition in detail to show how it works. The following appendix gives some useful pointers about working with the tree style.

Here's the extended tree-style definition example from this guide:

```
job
   .SetOptimizationMethod(tsopt.OptimizationMethod.Genetic) ①
   .Securities                                              ②
      .AddSecurity()                                        ③
         .SetSymbol("CSCO")                                 ④
         .Interval                                          ⑤
            .SetMinuteChart(5)
         .EndInterval                                       ⑥
         .History                                           ⑦
            .SetLastDate("12/31/2012")
            .SetDaysBack(180)
         .EndHistory                                        ⑧
      .EndSecurity                                          ⑨
   .EndSecurities                                           ⑩
   .Strategies                                              ⑪
      .AddStrategy("Bollinger Bands LE")                    ⑫
         .ELInputs                                          ⑬
            .OptRange("Length", 10, 30, 1
            .OptRange("NumDevsDn", 1, 3, 0.1)
         .EndELInputs                                       ⑭
      .EndStrategy
      .AddStrategy("Bollinger Bands SE")                    ⑮
         .ELInputs
            .OptRange("Length", 10, 30, 1)
            .OptRange("NumDevsUp", 1, 3, 0.1)
         .EndELInputs
      .EndStrategy
   .EndStrategies                                           ⑯
   .Settings                                                ⑰
      .GeneticOptions                                       ⑱
         .SetPopulationSize(200)
         .SetGenerations(75)
      .EndGeneticOptions
      .ResultOptions                                        ⑲
         .SetFitnessMetric(tsopt.MetricID.TSIndex)
         .SetNumTestsToKeep(300)
      .EndResultOptions
   .EndSettings
.UseDefinition();                                           ⑳
```

Here's a detailed explanation of how the example works:

1. The `SetOptimizationMethod` call returns a reference to the `Job` object, so that we can call another method on the job.

2. The `Securities` property returns a `Securities` object, which provides methods to add, get, or delete Security nodes.
3. The `Securities.AddSecurity` method adds a security and returns a `Security` object that represents that node. This object provides methods to define a security.
4. The `Security.SetSymbol` method sets the symbol and returns a reference to the `Security` object so that we can call another method or property on the security.
5. The `Security.Interval` property returns an `Interval` object, which provides methods to set different kinds of intervals. We call the `SetMinuteChart` method to define the interval.
6. The `EndInterval` property returns the parent object of the `Interval`, which is a `Security` object.
7. We can now access `History`, which is another property of the `Security` object. This provides various methods to set the history range. Here we call the `SetLastDate` and `SetDaysBack` methods.
8. Then we can call `EndHistory` to return to the `Security` object again.
9. We've now defined all the key parts of a security (symbol, interval, and history), so we call `EndSecurity` to get back to the `Securities` object.
10. We don't need to add a second security for this job, so we call `EndSecurities` to get back to the `Job` object.
11. Since we are in the context of the `Job` object again, we can now build up a strategies sub-tree using the same basic techniques. We start by using the `Job.Strategies` property to get a `Strategies` object.
12. The `Strategies.AddStrategy` method adds a strategy with the specified name and returns a `Strategy` object, which provides properties and methods to define information about the strategy.
13. The `Strategy.ELInputs` property returns an `ELInputs` object for the strategy. This object provides various methods for either setting or optimizing inputs. Here we call the `OptRange` method twice to optimize two different inputs. Note that the `OptRange` method returns the `ELInputs` object so that we can call additional methods on it.
14. The `EndELInputs` property returns the parent object of the `ELInputs` object, which is a `Strategy` object. At this point we could define additional information about the strategy, such as intrabar order options or signal states. However, we don't need to do that in this example, so we close off the strategy definition with the `EndStrategy` property, which returns the parent `Strategies` object.
15. We repeat this process to add a second strategy and optimize two of its inputs.
16. After adding the two strategy definitions, we use the `Strategies.EndStrategies` property to get back to the `Job` object.
17. Now we want to change a few default settings, so we use the `Job.Settings` property to get the `Settings` object.
18. The settings are divided into groups of related options. Each group is represented by a sub-options object that can be accessed by a property with the same name. Here we use the `Settings.GeneticOptions` property to get a `GeneticOptions` object. Then we call a

couple `Set…` methods to set some options. Each of these methods returns the `GeneticOptions` object so that we can set additional options. Finally, we use the `EndGeneticOptions` property to get back to the `Settings` object.

19. We repeat this process to change a couple of the result options. Then we call `EndResultOptions` to get back to the `Settings` object and `EndSettings` to return to the `Job` object.
20. At the end of the definition, we must call the `UseDefinition()` method to complete the definition. The `EndSettings` member is a property, and EasyLanguage does not allow you to end a statement with a property (unless you are assigning the result of the whole definition to a variable). However, calling `UseDefinition()` at the end turns the definition into a valid EasyLanguage statement.

Appendix 2: Working with the Tree Style

If you are interested in trying the tree style of job definitions, here are some tips to get you started:

- When defining a complete job in the tree style, put each property or method call on a separate line. The job variable is usually placed by itself on the first line.
- Start each line of the definition except the first with a dot operator (the period key). The auto-complete feature will pop up a list of the available methods and properties for the current part of the tree. This helps you type the job definition quickly and correctly.
- To move to the next level in the tree, press the Tab key and then the period key.
- To move to the previous level in the tree, press the Shift+Tab key and then the period key.
- Do not use a setter property (a property followed by an equals sign) in a tree-style definition. Instead, you must use the `Set…` method for the property. Every writable property has a corresponding `Set…` method that sets the property and returns its object. This allows you to chain the method calls together. For example, you cannot do this in a tree-style definition:

```
job
   .Settings
      .GeneticOptions
         .PopulationSize = 200  //WRONG!!!
         .Generations = 75      //WRONG!!!
         etc.
```

  Instead, you must do this:

```
job
   .Settings
      .GeneticOptions
         .SetPopulationSize(200)
         .SetGenerations(75)
         etc.
```

  This works because the `SetPopulationSize` and `SetGenerations` methods both return the `GeneticOptions` object, so you can chain the methods together.

- The only properties used in a tree-style definition are read-only properties that return objects. For example, the `Settings` property in the example above returns the `Settings` sub-object of the `Job` object, and the `GeneticOptions` property returns the `GeneticOptions` sub-object of the `Settings` object. These properties allow you to navigate to a deeper level in the tree. Thus, you should press the Tab key on the following line to move to the next level.
- The `End…` properties are also read-only properties that return objects. These properties always return the parent object, so they allow you to navigate to the previous level in the tree. Thus, you should press the Shift+Tab key to move to the previous level *before* typing the dot operator and the `End…` property.
- Some methods also return sub-objects. For example, the `AddSecurity` method creates and returns a `Security` sub-object, and the `AddStrategy` method creates and returns a `Strategy` sub-object. Since you will typically call methods on the sub-objects to define them, you should press the Tab key on the following line to move to the next level.

- Remember that you cannot end a tree-style definition with an `End…` property. If the last statement in your definition is an `End…` property, add a `UseDefinition()` call to complete the definition. On the other hand, if you have written a partial job definition that just chains some `Add…` calls together, you can omit both the `End…` property and the `UseDefinition()` call.

To sum up, a tree style definition uses the following kinds of methods and properties:

- Read-only properties that return a sub-object. These always require additional information, so press the Tab key on the following line to indicate that you are "inside" the object.
- `End…` properties that return the parent object. Press Shift+Tab to return to the previous level before typing the property.
- `Add…` methods that add a sub-object. If the sub-object requires additional information, press the Tab key on the following line to indicate that you are "inside" the object.
- `Add…` methods that add a value or string (e.g. `AddValue`). These just return the method's object so that you can chain the calls together. Don't change the indentation until the next `End…` property.
- `Set…` methods that set a property on the object. These just return the object so that you can chain the calls together. Don't change the indentation until the next `End…` property.
- `Opt…` methods that optimize a parameter. Most of these just return the method's object so that you can chain the methods together at the same level. However, the `OptList` method requires additional information (the values to optimize), so it returns an `OptList` sub-object.

Note: `OptSymbol` and `OptInterval` are defined as *properties* of the `Security` object, so they fall under the first category above: read-only properties that return a sub-object. They are defined this way because they don't require an argument (unlike `OptList`, which needs the input name), and declaring them as properties simplifies the syntax in several situations. (For example, this allows you to index directly into an `OptSymbol` or `OptInterval` property when you are querying an existing job definition.)

After you gain some experience working with the tree style, writing a job definition can become a very fluid process. The indentation helps you keep track of where you are and what you need to do next, and auto-complete allows you to type the methods and properties quickly and accurately.