EasyLanguage Optimization API 開発者ガイド

2016 年 1 月 8 日

TradeStation®

重要な情報:

TradeStation 系列会社は、いかなる証券、証券デリバティブ、先物商品、または取引所外外国為替取引 (FX) あるいは各種の取引または投資に関するアドバイス、推薦、またはストラテジーを売買する提案または勧誘を行わず、いかなる方法でも推奨しません。また、本 Web サイトで入手できる情報は、TradeStation が取引を許可されていない地域においては、いかなる種類の提案または推薦を意味するものではありません。この地域には日本が含まれますが、日本に限定されません。

過去のパフォーマンスは、それが実際のものか、ストラテジーのヒストリカルテストにより示されたものかにかかわらず、将来のパフォーマンスまたは成功を保証するものではありません。
取引する資産クラス (株式、オプション、先物、または FX) を問わず、すべての投資またはそれ以上の損失を被る可能性があります。したがって、損失を許容できない資産は投資しないでください。
オプション取引に向かない投資家もいます。オプションを取引するための口座アプリケーションは、取引経験を含む、すべての関連要因に基づいて検討され、認可または不認可になります。
ここをクリックして、「Characteristics and Risks of Standardized Options」という文書を確認してください。
資産クラスの取引前に、顧客は「Other Information」ページの関連するリスク開示説明書を読む必要があります。市場のボラティリティーや出来高、相場の遅延、システムやソフトウェアのエラー、インターネットのトラフィック、停電などの要因により、システムアクセスおよび取引発注と執行が遅延するか失敗する場合があります。

TradeStation Group, Inc. 系列会社: TradeStation の特許技術はすべて、TradeStation Technologies, Inc. が所有します。株式、株式オプション、および商品先物の商品とサービスは、TradeStation Securities, Inc. (NYSE、FINRA、NFA、および SIPC のメンバー) により提供されます。

TradeStation Securities, Inc. の SIPC は、株式と株式オプションの口座のみを対象にしています。
FX 商品とサービスは、IBFX, Inc. (NFA のメンバー) の 1 部門である TradeStation Forex により提供されます。

目次

はじめに 1
単純な最適化アプリケーション 1
Optimization API の概要 3
tsopt.Job クラス 4
tsopt.Optimizer クラス 4
tsopt.ProgressInfo クラス 4
tsopt.BestValues クラス 4
tsopt.Results クラス 4
tsopt.OptimizationException クラス 4
最適化ジョブの定義 5
ジョブ定義の構造 5
従来形式でのジョブの定義 6
ツリー形式でのジョブの定義 8
漸増的なジョブの定義 10
ストラテジーパラメーターの最適化 11
範囲による最適化 12
リストによる最適化 12
Boolean 入力の最適化 13
固定値への入力の設定 13
ストラテジーの有効なステータスの最適化 14
証券パラメーターの最適化 14
シンボルの最適化 14
足種の最適化 15
シンボル、足種、入力の複合最適化 16
ジョブ設定の指定 18
最適化の実行 20
最適化ジョブの定義 20
イベントハンドラーの定義 20
ProgressChanged イベント 20
JobFailed イベント 21
JobDone イベント 22

最適化の開始 ... 23
最適化のキャンセル ... 24
エラー処理 ... 26
ジョブ定義のエラーの捕捉 ... 26
検証エラーの処理 ... 27
実行時エラーの処理 ... 27
エラー処理の概要 ... 28
最適化結果の取得 ... 28
ストラテジーメトリクスと tsopt.MetricID ... 28
結果の構造 ... 28
テストに関する情報の取得 ... 29
ストラテジーメトリクスの取得 ... 30
第 1 テストの名前付きメソッドの使用 ... 30
特定のテスト、取引タイプ、範囲の名前付きメソッドの使用 ... 30
GetMetric Method の使用 ... 30
最適化されたパラメーターデータの取得 ... 31
特定の種類のパラメーターデータの取得 ... 31
結果のソート ... 33
例: テキストファイルへの結果の書き込み ... 33
簡単な方法での結果の書き込み ... 35
キューに置かれる最適化 ... 36
ジョブ ID ... 36
キューに置かれた最適化のキャンセル ... 38
API Reference ... 39
Job Definition Classes ... 39
DefinitionObject Class ... 39
Job Class ... 39
Securities Class ... 40
Security Class ... 41
Interval Class ... 44
History Class ... 47
OptSymbol Class ... 51
OptInterval Class ... 52
SecurityOptions Class ... 55

Strategies Class 56
Strategy Class 57
ELInputs Class 58
ELInput Class 60
OptRange Class 62
OptList Class 63
SignalStates Class 65
IntrabarOrders Class 66
Settings Class 67
GeneticOptions Class 68
ResultOptions Class 69
GeneralOptions Class 70
CostsAndCapital Class 71
PositionOptions Class 73
TradeSize Class 73
BackTesting Class 74
OutSample Class 75
WalkForward Class 77
CommissionBySymbol Class 78
SlippageBySymbol Class 80
AvailableStrategies Helper Class 83
AvailableSessions Helper Class 84
Optimization Classes 86
Optimizer Class 86
JobDoneEventArgs Class 89
JobFailedEventArgs Class 89
ProgressChangedEventArgs Class 90
ProgressInfo Class 90
BestValues Class 91
OptimizationException Class 92
The Results Class 94
Appendix 1: Anatomy of a Tree-Style Definition 103
Appendix 2: Working with the Tree Style 105

EasyLanguage Optimization API

はじめに

EasyLanguage Optimization API は、EasyLanguage を使用する最適化ジョブを定義し実行するのに必要なすべてのツールを提供します。以下のように、最適化のあらゆる面を制御できるようになります。

- シンボル、足種、履歴範囲を含む、最適化に使用する証券を定義します。複数のシンボルまたは足種を最適化することも可能です。
- 最適化に使用するストラテジーを定義します。
- 最適化する入力を指定します。値の範囲だけでなく、値や式のリストも最適化できるようになります。
- ストラテジー固有の設定の他に、最適化用のグローバル設定を定義します。
- 実行時に最適化の進捗をモニターします。
- 最適化の完了時に最適化結果を取得し分析します。
- 最適化のエラーを処理します (発生した場合)。

最適化を完全にプログラムできるため、複数の最適化を自動で実行するアプリケーションを作成することもできます。実際に、ある最適化の結果を使用して、次回の最適化のパラメーターを決定することができます。これにより、ストラテジー分析の新しい可能性が開かれます。

Optimization API は、マルチスレッドの最適化エンジンも最大限に活用します。これには手間をかける必要はありません。最適化する対象とイベントへの対処方法を指定するだけで、コンピューターのコアをすべて自動的に使用できるようになります(必要に応じて、最適化に使用するスレッド数も指定できます)。

単純な最適化アプリケーション

API の詳細を掘り下げる前に、EasyLanguage で作成された単純な最適化アプリケーションをここで紹介します。コードを読み、API の特長をつかんでください。次のセクションでは、アプリケーションが使用するテクニックについて深く掘り下げます。

この TradingApp は、入力で指定されたシンボル、分足、履歴範囲の Bollinger Bands LE および SE ストラテジーを最適化します。アプリケーションは、各ストラテジー (NumDevsUp と NumDevsDn) の偏差数を最適化します。最適化に使用するステップサイズも入力として指定されます。

アプリケーションは、実行時にその進捗を Print Log に出力し、最適化の完了時に最良の Net Profit を出力します。

このコードに自分で入力する場合は、必ず `InitApp` メソッドを TradingApp の `Initialized` イベントに割り当ててください。

コンピューターに複数のコアがある場合、このアプリケーションはマルチスレッドの最適化を自動で実行します。

```
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 の概要

最適化クラスはすべて、tsopt 名前空間内にあります。名前空間識別子はとても短く、クラスは他の識別子と競合する可能性がある単純な名前であるため、*名前空間のインポートは推奨**されません***。コードを作成する際は、tsopt. と入力して、オートコンプリートが提供するポップアップリストからクラスを選択するだけで、簡単にすばやく作成できます。同様に、Optimization API に含まれるクラスを簡単に見つけることもできます。

本ガイドのコード例全体を通じて、クラス名に名前空間修飾子を含めています。これにより、そのクラスが最適化に関連していることが明確になります。多くの場合、クラスの名前をとって変数に名前を付けますが、名前空間修飾子はありません。コード内でもこの規則は便利です。たとえば、次のコードスニペットは、optimizer という変数を tsopt.Optimizer クラスのインスタンスとして宣言しています。

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

ここでは、Optimization API の主なクラスの概要を簡単に示します。

tsopt.Job クラス

tsopt.Job クラスにより、クライアントアプリケーションは最適化ジョブを定義できます。この定義には、最適化に使用される証券、最適化されるストラテジー、および関連する設定が含まれます。

ジョブ定義は常に Job クラスから始まり、ジョブのさまざまな部分を記述するヘルパークラスも多数あります。たとえば、Job オブジェクトは、最適化に使用されるストラテジーを定義する tsopt.Strategies オブジェクトを含みます。Strategies オブジェクトは tsopt.Strategy オブジェクトのコレクションであり、それぞれがストラテジーを定義します。さらに Strategy オブジェクトには、ストラテジー入力を記述する tsopt.ELInputs オブジェクトなどがあります。ジョブ定義の構造については、以下に詳しく説明します。

tsopt.Optimizer クラス

tsopt.Optimizer クラスは、最適化ジョブを開始またはキャンセルするためのメソッドを提供します。このクラスは、最適化用のイベントハンドラーを追加する場所でもあります。

tsopt.ProgressInfo クラス

tsopt.ProgressInfo クラスは、現在のテスト番号、経過時間と残り時間、遺伝的最適化の場合は現在の世代および個体など、最適化の進捗に関する情報を提供します。ProgressInfo オブジェクトは、ProgressChanged イベントハンドラーの ProgressChangedEventArgs 引数から取得できます。

tsopt.BestValues クラス

tsopt.BestValues クラスは、最適化中にこれまでに見つかった最適な結果に関する情報を、その結果に関連付けられた最適化パラメーターとともに提供します。ProgressInfo 同様、このオブジェクトは、ProgressChanged イベントハンドラーの ProgressChangedEventArgs 引数から取得できます。

tsopt.Results クラス

tsopt.Results クラスは、最適化の結果を保持します。JobDone イベントハンドラーの JobDoneEventArgs 引数から取得できます。Results クラスは、結果セット内の最適化テストごとにストラテジーメトリクスを取得するメソッドを提供します。

tsopt.OptimizationException クラス

tsopt.OptimizationException クラスは、ジョブ定義の構築中にスローされる例外です。定義プロセス内のある種のコーディングエラー (空の Strategies オブジェクトへのインデックス付けなど) をデバッグする目的で、この例外を捕捉できます。Optimization API は、最適化の開始時にジョブ定義を確認して、ジョブ定義が正しく完全であることを保証します。検証エラーがあった場合、ライブラリーは tsopt.OptimizationException をスローします。

Optimization API は、最適化の*実行中*には例外をスローしないことに注意してください。その代わりに、エラーは `JobFailed` イベントハンドラーの呼び出しによりレポートされます。最適化は非同期で実行されるため、このテクニックは、最適化中に発生するエラーのレポートに最適です。

エラー処理については、以下で詳しく説明します。

最適化ジョブの定義

このセクションでは、ジョブ定義 API の概要を説明します。各ジョブ定義クラスの詳細については、本文書の後半にある「*API リファレンス*」を参照してください。

ジョブ定義の構造

ジョブ定義は、Securities、Strategies、および Settings という 3 つの主要なノードを有するツリーです。

Securities ノードは、最適化で使用する証券を定義する Security ノードを 1 つ以上含む必要があります。各 Security ノードは異なるデータシリーズ (TradeStation 内の Data1、Data2、Data3 など) に対応します。最適化の多くは、Security を 1 つだけ使用します。

Strategies ノードは、最適化で使用するストラテジーを定義する Strategy ノードを 1 つ以上含む必要があります。特定のストラテジーの入力と設定は、その Strategy ノード内で指定されます。

Settings ノードは、最適化用のさまざまなグローバル設定を定義するノードを 1 つ以上含みます。デフォルト設定が使用される場合には、Settings ノードを空にすることもできます。

ジョブ定義は、使用する **OptimizationMethod** も指定できます (Exhaustive または Genetic のいずれか)。最適化メソッドが指定されていない場合、デフォルトの Exhaustive に設定されます。

各 **Security** ノードは、証券のシンボル、足種、履歴範囲についての情報を含む必要があります。この情報のいずれかがない場合、ジョブの開始時に、オプティマイザーは例外をスローします。

Strategy ノードには、ストラテジー名が 1 つ必要です。他の情報が提供されていない場合、ストラテジーのデフォルトの入力と設定が使用されます。ただし、ほとんどのジョブ定義は、ストラテジーの追加情報を提供します。

ジョブ定義は、Strategy または Security 内のあるパラメーターを最適化するよう指定できます。たとえば、よくある事例として、ストラテジー内の入力を 1 つ以上最適化します。Optimization API では、通常これは、該当する入力の `OptRange` メソッドまたは `OptList` メソッドを呼び出すことにより指定されます。証券の一部も最適化できます。たとえば、`OptSymbol` プロパティを有するシンボルのリストを最適化することも、`OptInterval` プロパティを有する足種のリストを最適化することもできます。

ジョブ定義を作成するには、最初に `tsopt.Job` オブジェクトを作成する必要があります。

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

   // Define the job here

   return job;
end;
```

ジョブを作成した後、Job オブジェクトでプロパティとメソッドを呼び出して、ジョブのさまざまな部分を定義できます。次のセクションでは、ジョブを指定するための 2 種類のアプローチについて説明します。これは、(1) 従来の形式でジョブを定義する、または (2)「ツリー形式」でジョブを定義するというアプローチです。単一のアプリケーションで、これらのアプローチのいずれかまたは両方を使用できます。

ジョブ定義は基本的に*宣言*であることに注意してください。ジョブのコンテンツを*宣言する*一連のプロパティ呼び出しとメソッド呼び出しで構成されますが、ジョブ定義自体は実際に何かを*行う*わけではありません。ジョブを定義した後で、そのジョブを実行するためには定義を tsopt.Optimizer.StartJob メソッドに渡す必要があります。

ジョブ定義は宣言であるため、別の宣言形式で XML 間で容易に変換できます。

従来形式でのジョブの定義

各ステートメントがオブジェクトを作成する、オブジェクトでプロパティを設定する、またはオブジェクトでメソッドを呼び出す従来形式のコーディングを使用してジョブ定義を作成できます。

この形式を使用する場合、作成するさまざまなサブオブジェクトを保持するローカル変数をいくつか定義する必要があります(ローカル変数なしで従来形式のコードを作成できますが、非常に反復が多く、読みにくいコードになります)。

最低 1 つの証券とストラテジーを定義に必ず追加する必要があるため、少なくとも tsopt.Security オブジェクトと tsopt.Strategy オブジェクトのローカル変数を定義する必要があります。

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

証券を定義に追加するには、ジョブから Securities オブジェクトを取得し、その AddSecurity メソッドを呼び出して、その結果をローカル変数に割り当てます。

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

次に、証券のシンボル、足種、および履歴範囲を定義します。

ストラテジーを定義に追加するには、ジョブから Strategies オブジェクトを取得し、その AddStrategy メソッドを呼び出して、その結果をローカル変数に割り当てます。

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

次に、最適化する入力など、ストラテジーの情報を定義します。

定義に追加したいストラテジーごとに、このパターンを繰り返すことができます。追加するストラテジーごとに同じローカル変数を再利用しても構いません。

ジョブ定義に設定を追加する場合、設定のグループごとにローカル変数を定義することもできます。これは必須ではありませんが、コードの反復が少なくなります。たとえば、いくつかの遺伝オプションを変更する場合、GeneticOptions オブジェクトのローカル変数を作成できます。

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

次に、ジョブ定義の GeneticOptions オブジェクトをこの変数に割り当て、変数を使ってプロパティを設定できるようにします。

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

次の例は、従来形式で最適化ジョブを定義するメソッドを示しています。

```
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;
```

ツリー形式でのジョブの定義

特にアプリケーションのプロトタイピングやテストのとき、単一のステートメントでジョブ全体を定義すると非常に便利です。メソッド呼び出しを連鎖できるため、Optimization API はこの形式のジョブ定義をサポートしています。このようにジョブを定義するときは、インデントを使用して、ツリー構造を強調表示すると効果的です。そのため、これを「ツリー形式」のジョブ定義と呼びます。

これは例示するのが一番であると思われます。

```
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();
```

ツリー形式には便利な機能がいくつかあります。

- この形式では、構文と書式設定でツリー構造のジョブ定義が強調されるため、ジョブの全体構造を把握しやすくなります。
- ローカル変数を定義する必要はありません。ツリーにサブオブジェクトを追加したとき、またはツリー内のサブオブジェクトにアクセスしたとき、関連メソッドまたはプロパティをすぐに呼び出して、そのオブジェクトを定義できます。たとえば、`AddSecurity` メソッドは空の `Security` オブジェクトを返します。そのオブジェクトを変数に保存するのではなく、返されたオブジェクトで直接メソッドまたはプロパティを呼び出すだけで、証券を定義できます。証券の複数の側面を定義するには、呼び出しを連鎖します。
- 特定のオブジェクトの「内部」にいるとき、その `End...` プロパティを使用して、親オブジェクトのコンテキストに戻ることができます。たとえば、`Interval` オブジェクトの `EndInterval` プロパティを使用して、親の `Security` オブジェクトに戻ることができます。また、`Security` オブジェクトの `EndSecurity` プロパティを使用して、親の `Securities` オブジェクトに戻ることができます。これにより、ジョブの定義中にツリーを「移動」できます。
- ツリー形式には反復がほとんどありません。オブジェクトの「内部」に入ると、メソッドとプロパティを連鎖するだけで、目的のオプションをすべて定義できます。従来形式では必要となることが多い、コードの各行で受信側オブジェクト (またはオブジェクトの連鎖) を繰り返す必要がありません。これは、多数のパラメーターを最適化する、または多数のデフォルト設定を変更する定義で特に役立ちます。
- オートコンプリートはツリー形式での使用に適しています。各行の始めにドット演算子を入力すると、オートコンプリートは、ツリーの現在の部分に関連するメソッドとプロパティのリストをポップアップ表示します。たとえば、`History` オブジェクトの内部にいる場合、履歴範囲を設定するためのメソッドがすべて表示されます。ほとんどの定義メソッドは、「Set…」(プロパティを設定する)、「Add…」(オブジェクトまたは値を追加する)、または「Opt…」(パラメーターを最適化する) で始まります。

ツリー形式は完全に任意です。従来形式でジョブ定義をコーディングしやすければ、それで構いません。その場合でも、Optimization API のすべての機能にアクセスできます。とはいえ、XML などのツリーベースの形式に慣れている場合は、ツリー形式を試してみようと思うかもしれません。コツがわかれば、とても効率的かつ直感的に最適化ジョブを定義できるようになります。

連鎖したメソッドでジョブを定義するとき、上記の例のインデント形式を使用すると便利です。コードの動作には必要ありませんが、ジョブ定義の構造が把握しやすくなります。TradeStation Development Environment では、Tab キーを押して次のインデントレベルに移動したり、Shift+Tab キーを押して前のインデントレベルに戻ったりすることができるため、このインデント形式でのコードの作成がシンプルになります。さらに、各行をドットで開始すると、オートコンプリートにより、ジョブツリーの現在の部分に使用できるメソッドとプロパティがポップアップ表示されます。(Optimization API で作業をする際は、オートコンプリートを有効にしておくことを強くお勧めします。)

ツリー形式の定義の最後を `End...` プロパティにすることはできません。EasyLanguage では、ステートメントの最後をプロパティにすることはできないためです (割り当ての一部である場合を除く)。そのため、定義の最後の部分が `End...` プロパティになる場合は、`UseDefinition()` メソッド呼び出しで定義を終わる必要があります。これにより、ツリー定義が有効な EasyLanguage ステートメントになります。「An

equals sign '=' expected here」という構文エラーが表示された場合は、UseDefinition() に呼び出しを追加すると、エラーが修正されます。

ツリー形式の機能の詳細については、「*付録 1: ツリー形式の定義の構造*」を参照してください。ツリー形式の使用に関する有益な指針については、「*付録 2: ツリー形式の使用*」を参照してください。

漸増的なジョブの定義

UI ベースの最適化アプリケーションでは通常、ジョブ定義が全部一度に定義されることはありません。その代わり、ユーザーが提供した入力から漸増的に組み立てられます。Optimization API は、この形式のジョブ定義を完全にサポートします。実際、非常に柔軟性があるため、ユーザーが選択したオプションの単独の内部表現としてジョブ定義を使用できます。ユーザーの最適化選択を保持する別個のデータ構造を作成する必要がありません。データ構造を直接ジョブ定義に入れ、ユーザーの入力に基づき必要に応じてアップデートできます。

ジョブに対して増分アップデートを行うには、tsopt.Job オブジェクトから始めて、適切なメソッドやプロパティを呼び出し、ジョブツリーの目的の部分に移動します。呼び出しを連鎖して、コードを簡潔にしておくことができます。たとえば、5 分間隔から 10 分間隔に変更するとします。Job オブジェクトが「job」という名前の場合、次のように実行します。

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

オブジェクトのコレクションを返す Securities のようなプロパティの場合、プロパティにインデックス付けして、そのサブオブジェクトの 1 つを取得できます。上記の例では、Securities[0] を使用して、ジョブ定義にある最初の Security オブジェクトを取得します。その後、Interval を呼び出して、その Interval オブジェクトを取得します。最後に、SetMinuteChart を呼び出して間隔を設定します。

Job オブジェクトを初めて作成すると、Securities、Strategies、および Settings のノード以外は完全に空のオブジェクトになります。そのため、適切な Add... メソッドを呼び出して、証券とストラテジーをジョブに追加する必要があります。(新たに作成したジョブで上記のコード例を試すと、エラーになります。これは Securities コレクションが空であるためです。)どの最適化ジョブも証券を少なくとも 1 つ含む必要があるため、Job オブジェクトの作成直後に Security オブジェクトを追加できます。

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

さらに、ユーザーが変更できるデフォルトの証券も提供できます。

```
job = new tsopt.Job;

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

ジョブ定義ツリーの特定の場所で複数の変更を行う必要がある場合、ツリーのその部分に移動して、オブジェクトをローカル変数に保存することをお勧めします。その後、そのオブジェクトで呼び出しを繰り

返して、変更を適用します。これは、変更ごとにオブジェクトに移動するよりも効率的です。

たとえば、最初のストラテジーの Length 入力を最適化し、その入力で離散値のリストをテストするとします。Optimization API は、OptList プロパティを介してこの種類の最適化をサポートします。入力値のリストが values という double 型の Vector に含まれている場合、次のようにこの最適化を指定できます。

```
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;
```

ループの反復ごとに Job オブジェクトから OptList オブジェクトに移動することは非効率的です。ローカル変数に OptList オブジェクトを保存すると、この問題が解決されるだけでなく、コードが読みやすくなります。

ストラテジーパラメーターの最適化

Optimization API は、ストラテジーを最適化するさまざまな方法を提供します。

1. 数値の範囲で入力を最適化できます。これは、TradeStation がサポートする従来の方法です。
2. 値のリストで入力を最適化できます。これは、次のシナリオに役立ちます。
   a) 範囲で表すことができない離散的な値のセット (1、3、7、15、30 など) を最適化する場合。
   b) 異なるテキスト値のリストを渡して、テキスト入力を最適化する場合。
   c) 例外が予想される入力を最適化する場合 (BollingerPrice など)。リストを使用して、その入力に対して複数の式をテストできます (「終値」、「高値」、および「安値」など)。
3. True 値および False 値をテストして Boolean 入力を最適化できます。
4. ストラテジーが有効化されているかどうかを最適化できます。これは、協同ストラテジーのセットを最適化していて、ストラテジーが有益な貢献をするかどうかを判断する目的で、ストラテジーの一部をオンまたはオフにするときに役立ちます。

次のセクションでは、Optimization API を介してこれらのさまざまな最適化をリクエストする方法について説明します。

範囲による最適化

数値の範囲で入力を最適化するには、ELInputs.OptRange メソッドを呼び出し、入力の名前、範囲の始まり、範囲の終わり、および増分値を渡します。たとえば、次のコードを使用して、最初のストラテジーの Length 入力を 10 から 30 まで 2 ずつ最適化します。

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

リストによる最適化

値のリストで入力を最適化するには、ELInputs.OptList メソッドを呼び出し、入力の名前を渡します。メソッドは、OptList オブジェクトを返します。これは、値の追加、変更、または削除のメソッドを提供します。

従来形式を使用している場合、反復を避けるために OptList オブジェクトを変数に保存することをお勧めします。

```
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;
```

連鎖メソッド形式を使用している場合、AddValue メソッドを連鎖して、リストに入力できます。

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

複数のメソッド呼び出しを連鎖していて、そのメソッド呼び出しの後に End… プロパティを続けていない場合、末尾の UseDefinition() 呼び出しを省略できることに注意してください。

ツリー形式の定義内では、EndOptList プロパティを使用して、ELInputs オブジェクトのコンテキストに戻ることができます。これにより、追加の ELInputs メソッドを連鎖できます。

```
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();
```

OptList で反復アップデートを実行するには、ローカル変数にオブジェクトを保存すると、効率良くアクセスできます。

```
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;
```

数値または式のいずれか (あるいは両方) を OptList に追加できることに注意してください。数値をリストに追加するには、AddValue メソッドを呼び出し、数値を渡します。式をリストに追加するには、AddValue メソッドを呼び出し、式を引用符で囲みます。

注: 数値を含む文字列は有効な式であるため、これを渡すこともできます。したがって、TextBox から値を取得する場合、その値を OptList オブジェクトに追加する前に数値に変換する必要はありません。AddValue("15") の呼び出しと AddValue(15) の呼び出しに違いはありません。

テキスト値のリストを最適化することもできます。これは明らかに特殊なケースですが、そうしたい場合は、文字列がテキスト値であり式ではないことを示すために、AddValueAsText メソッドを使用できます。たとえば、「MethodName」テキスト入力を含む「Method Test」というストラテジーがあるとします。ストラテジーは、入力を介して渡されたメソッドの名前に基づいてさまざまな操作を実行します。次のように、メソッド名のリストを最適化できます

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

OptList への省略可能な第 2 引数で、入力タイプが TextType であることを指定する必要もあることに注意してください。デフォルトの入力タイプは NumericType であるため、数値入力の場合は省略できます。

Boolean 入力の最適化

Boolean 入力を最適化するには、ELInputs.OptBool メソッドを呼び出し、入力の名前を渡します。オプティマイザーは入力の True 値と False 値のみをテストするため、追加の情報を提供する必要はありません。

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

固定値への入力の設定

ジョブ定義で入力を固定値に設定するには、ELInputs.SetInput メソッドを呼び出し、入力の名

前、値、および任意でタイプ (デフォルトは Numeric) を渡します。OptList と同様に、入力の数値、数式、またはテキスト値を渡すことができます。

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

テキスト値を指定するには、SetInput メソッドではなく SetInputAsText メソッドを呼び出します。

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

入力を最適化しない、または明示的に値を設定する場合、Optimization API は入力のデフォルト値を使用します。

ストラテジーの有効なステータスの最適化

協同ストラテジーのグループがある場合、ストラテジーが有益な貢献をするかどうかを判断するために、1 つ以上のストラテジーの Enabled ステータスを最適化できます。ストラテジーの Strategy.OptStrategyEnabled メソッドを呼び出します。たとえば、次のコードは、第 3 ストラテジーの Enabled ステータスを最適化します。

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

メソッドは、ストラテジー全体に適用されるため、ELInputs サブオブジェクトではなく Strategy オブジェクトで呼び出されます。

証券パラメーターの最適化

Optimization API では、証券を最適化することもできます。任意の証券 (データシリーズ) のシンボルや足種を最適化できます。単一のジョブ内に証券の最適化とストラテジーの最適化を組み合わせることもできます。

シンボルの最適化

シンボルを最適化するには、SetSymbol ではなく Security.OptSymbol プロパティを使用します。返される OptSymbol オブジェクトは、テストするシンボルを追加する AddSymbol メソッドを提供します。

従来形式を使用している場合、OptSymbol オブジェクトを変数に保存することをお勧めします。次に、AddSymbol を複数回呼び出して、オブジェクトにシンボルを追加します。オプティマイザーは、他の最適化パラメーターと組み合わせて、各シンボルをテストします。たとえば、次のコードは最初の証券をアップデートして、3 つの異なるシンボルを最適化します。

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

ツリー形式を使用している場合は、AddSymbol 呼び出しを連鎖できます。

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

シンボルが配列またはコレクションに格納される場合、効率を良くするために、OptSymbol オブジェクトをローカル変数に保存します。

```
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;
```

ツリー形式のジョブ定義では、証券の別の部分を定義できるように、EndOptSymbol プロパティを使用して Security オブジェクトのコンテキストに戻ることができます。

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

足種の最適化

足種を最適化するには、Interval ではなく Security.OptInterval プロパティを使用します。返される OptInterval オブジェクトは、最適化でテストする異なる足種を追加するためのさまざまな Add… メソッドを提供します。分足やカギ足など、まったく異なる種類の足種を混在させることもできます。すべての足種が単一の履歴範囲で機能することを確認します。たとえば、通常、日足はティックよりもはるかに長い履歴範囲を必要とするため、ティックベースの足種と日足を混在させたくない場合があります。

従来形式を使用している場合、OptInterval オブジェクトを変数に保存することをお勧めします。次に、該当する Add… メソッドを複数回呼び出して、オブジェクトに足種を追加します。たとえば、次の

コードは最初の証券をアップデートして、いくつかの異なる分足を最適化します。

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

ツリー形式を使用している場合は、Add… メソッドを連鎖できます。

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

次の例は、異なる高度な足種をいくつか最適化します。

```
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);
```

足種を追加するためにループを使用している場合、効率を良くするために、OptInterval オブジェクトをローカル変数に保存します。たとえば、次のコードは、startMinutes、stopMinutes、および step 引数で指定された一連の分足を追加します。

```
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;
```

ツリー形式のジョブ定義では、証券の別の部分を定義できるように、EndOptInterval プロパティを使用して Security オブジェクトのコンテキストに戻ることができます。

シンボル、足種、入力の複合最適化

シンボル、足種、およびストラテジーを単一のジョブで最適化できます。オプティマイザーは、すべての最適化パラメーターのさまざまな組み合わせをテストします。(Exhaustive 最適化では、オプティマイザーは、シンボル、足種、および最適化入力のすべての可能な組み合わせをテストします。Genetic 最

適化では、オプティマイザーは、汎用アルゴリズムが提供する組み合わせをテストします。)

たとえば、次の完全なジョブ定義は、2 つのシンボル、2 つの足種、および 2 つのストラテジー入力を最適化します。

```
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;
```

以下は、同じジョブ定義をツリー形式にしたものです。

```
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();
```

ジョブ設定の指定

Optimization API は、TradeStation ストラテジーに利用可能な設定をすべてサポートします。これには、最適化ですべてのストラテジーに適用される設定だけでなく、特定のストラテジーに固有の設定も含まれます。

特定のストラテジーに適用される設定は、該当する `Strategy` ノード内で定義されます。すべてのストラテジー (または最適化自体) に適用される設定は、ジョブの `Settings` ノード内で定義されます。

ジョブ定義で設定を行っていない場合、その設定のデフォルト値が最適化において使用されます。

利用可能なグローバル設定が多数あるため、`Settings` オブジェクトはそれらをいくつかのカテゴリーにグループ化します。各カテゴリーは、異なるサブオプションオブジェクトで表され、同じ名前を有するプロパティでアクセスできます。Settings オブジェクトは、次のサブオプションプロパティを提供します。

- `GeneticOptions` (汎用最適化用)
- `ResultOptions` (保持するテストの数、適合度関数など)
- `GeneralOptions` (基準通貨、MaxBarsBack、Look-Inside-Bar など)
- `CostsAndCapital` (手数料、スリッページ、資本など)

- `PositionOptions` (ポジションあたりの買い増しオプションおよび最大株数)
- `TradeSize` (取引あたりの株数または通貨)
- `BackTesting` (転記オプション、マーケットスリッページなど)
- `OutSample` (アウトオブサンプル範囲の数量 (ある場合))
- `WalkForward` (ウォークフォワード最適化の有効化/無効化、ウォークフォワードテストの名前)

したがって、グローバル設定を変更するには、以下のステップを実行する必要があります。

1. `Job` オブジェクトから `Settings` オブジェクトを取得します。
2. `Settings` オブジェクトから該当するサブオプションオブジェクトを取得します。
3. サブオプションオブジェクト内の目的のプロパティを設定します。

多くの場合、単一のステートメントで最初の 2 ステップを実行して、ローカル変数にサブオプションオブジェクトを保存します。その後、そのオブジェクトで該当するプロパティを設定できます。

たとえば、汎用最適化を実行するジョブを定義してから、そのジョブの汎用オプションを変更するとします。`tsopt.GeneticOptions` タイプのローカル変数 `geneticOptions` を宣言していると仮定して、以下を実行します。

```
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;
```

プロパティを 1 つだけ設定している場合は、サブオプションオブジェクトをローカル変数に保存する必要はありません。その代わり、単一のステートメントでプロセス全体を実行できます。

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

複数のサブオプションの設定時にローカル変数を省略することもできますが、効率的ではなく、冗長になります。

オプションのグループを表す各オブジェクトは、各オプションの `Set…` メソッドも提供します。したがって、連鎖されたメソッドを使用して、複数のオプションを単一のステートメントで効率良く修正できます。

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

さらに、各オプションオブジェクトは、ジョブツリーのその親オブジェクトを返す `End…` プロパティを返します。これにより、複数のオプショングループを単一のステートメントで修正できます。

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

すべてのグローバル設定の詳細な説明については、「*API リファレンス*」の `tsopt.Settings` オブジェクトを参照してください。

ストラテジー固有設定の説明については、「*API リファレンス*」の `tsopt.Strategy` オブジェクトを参照してください。

最適化の実行

これまで、最適化ジョブを定義する方法を学習してきました。これで、最適化アプリケーションの他のコンポーネントを組み立てることができるはずです。

次のセクションでは、プロセスの各ステップについて詳しく説明します。

最適化ジョブの定義

最初に、前のセクションの説明に従って、最適化ジョブを定義するコードを作成します。必要に応じて、ジョブ定義全体をハードコーディングできますが、UI ベースのアプリケーションは、ユーザーが指定した入力に基づいて漸増的にジョブ定義を組み立てる可能性があります。

XML ファイルからジョブ定義をロードする可能性もあります。詳細については、「*API リファレンス*」の `Job` オブジェクトを参照してください。

イベントハンドラーの定義

Optimization API を使用して最適化を実行すると、独自のスレッド上で非同期で実行されます。したがって、API は、最適化の実行中にアプリケーションと通信し、最適化の完了時にアプリケーションに通知する方法を提供する必要があります。Optimization API は、`JobDone`、`JobFailed`、および `ProgressChanged` という 3 つのイベントを介してこの連絡を行います。アプリケーションは、最初の 2 つのイベントのイベントハンドラーを提供する必要があります。提供しないと、最適化の開始時にエラーとなります。`ProgressChanged` イベントのハンドラーは任意ですが、提供することをお勧めします。

次のセクションでは、各イベントの目的とそのハンドラーを実装する方法について説明します。

ProgressChanged イベント

最適化の実行中、Optimization API は `ProgressChanged` イベントを定期的に送信して、最適化の

進捗に関する情報を提供します。このイベントのハンドラーを提供すると、実行時に最適化の進捗を表示できます。

ProgressChanged イベントハンドラーには次のシグニチャーがあります。

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

ProgressChangedEventArgs クラスには、Progress と BestValues という 2 つのクラスがあります。

Progress プロパティは ProgressInfo オブジェクトを返します。これは、テスト番号、経過時間、および残り時間など、最適化の現在のステータスについての情報を提供します。汎用最適化の場合、現在の世代と個体についての情報も提供します。ProgressInfo クラスの詳細な情報については「API リファレンス」を参照してください。

BestValues プロパティは BestValues オブジェクトを返します。これは、これまでに見つかった最適な結果とその結果に関連付けられた最適化パラメーターを提供します。BestValues クラスの詳細な情報については「*API リファレンス*」を参照してください。

次に、ProgressChanged ハンドラーの簡単な実装を示します。メインフォームにはユーザーにステータスを表示する statusLabel という Label が含まれていると仮定します。このメソッドは、現在のテストとテストの数でステータスをアップデートします。

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

イベントハンドラーを定義した後、そのハンドラーを tsopt.Optimizer オブジェクト内のイベントに割り当てる必要があります。

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

JobFailed イベント

最適化中にエラーが発生すると、Optimization API は JobFailed イベントを送信します。このイベントのハンドラーを*必ず*提供してください。提供しない場合、アプリケーションは、いつ最適化がエラーで停止したかを知ることができません(バックグランドで非同期で動作するため、最適化の実行中、最適化エンジンはエラーに対して例外をスローしません)。JobFailed イベントハンドラーを提供し忘れた場合、最適化の開始時に API は InvalidOperationException をスローします。

JobFailed イベントハンドラーには次のシグニチャーがあります。

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

JobFailedEventArgs クラスには Error というプロパティがあります。これは tsopt.OptimizationException クラスのインスタンスを返します。そのオブジェクトで Message プロパティを呼び出して、エラーメッセージを取得できます。

次に、JobFailed ハンドラーの簡単な実装を示します。フォームには、statusLabel という Label が含まれていると仮定します。

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

必要であれば、Events Log でエラーを簡単にレポートできます。エラー情報は tsopt.OptimizationException オブジェクト内で提供されるため、イベントハンドラーから例外をスローできます。

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

イベントハンドラーを定義した後、そのハンドラーを tsopt.Optimizer オブジェクト内のイベントに割り当てる必要があります。

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

以下の「*エラー処理*」セクションでは、最適化エラーについて詳細に説明します。

JobDone イベント

最適化が正常に (つまり、エラーなしで) 停止すると、Optimization API は JobDone イベントハンドラーを呼び出します。イベントハンドラーが呼び出される理由は 2 つあります。

- 最適化が正常に完了した。
- 最適化がユーザーによりキャンセルされた。

アプリケーションが最適化をキャンセルする方法を備えている場合、このケースを検出して、適切なステータスメッセージを表示できます。あるいは、完了した最適化とキャンセルされた最適化を同じように扱うことができます。どちらのケースでも、最適化結果を取得してレポートできます。最適化がキャンセルされた場合、結果には完了したテストがすべて含まれます。

ほとんどのアプリケーションは、ユーザーが最適化をキャンセルしたかどうかの独自の記録を保持します (通常、上位の変数内)。その変数を確認して、キャンセルが発生したかどうかを判断できます。JobDoneEventArgs オブジェクトの Canceled プロパティを確認して、ユーザーが最適化をキャンセルしたかどうかを判断することもできます。

JobDone イベントハンドラーを*必ず*提供してください。提供しない場合、アプリケーションは、いつ最適化が終了したかを知ることができません。JobDone ハンドラーを提供し忘れた場合、最適化の開始時に API は InvalidOperationException をスローします。

JobDone イベントハンドラーには次のシグニチャーがあります。

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

一般的な JobDone イベントハンドラーは、次のタスクを実行します。

1. アプリケーションがキャンセルをオプションとして提供している場合、ステータス表示をアップデートして、最適化が完了したかキャンセルされたかを表示します。
2. args.Result プロパティから tsopt.Results オブジェクトを取得し、それを使用して最適化結果を表示または保存します。tsopt.Results クラスは、後のセクションで詳しく説明します。

次に、OptimizationDone メソッドの簡単な実装を示します。より詳細な実装はこの例と同じ基本パターンに従いますが、通常は最適化結果の詳細を提供します。

```
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;
```

イベントハンドラーを定義した後、そのハンドラーを tsopt.Optimizer オブジェクト内のイベントに割り当てる必要があります。

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

最適化の開始

ジョブの定義、tsopt.Optimizer オブジェクトの作成、およびイベントハンドラーの実装が完了したら、残りは最適化を開始するタスクだけです。

最適化を実行するには、Optimizer.StartJob メソッドを呼び出し、それにジョブ定義を渡します。第 2 引数として使用するスレッド数を渡すこともできます。省略した場合、Optimization API はプロセッサーのコア数に基づいてスレッド数を自動的に判断します。

通常、最適化は、ボタンのクリックまたはメニューの選択に応じて開始されます。TradingApp が optimizer という変数を含み、ジョブ定義が job という変数に格納されると仮定します。フォームに

StartButton というボタンがある場合、最適化を開始する次の Click ハンドラーを作成できます。

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

ジョブ定義にエラーがある場合、API は tsopt.OptimizationException をスローし、Events Log にエラーがレポートされます。自分でエラーを処理する場合、try/catch ブロックに StartJob 呼び出しをラップして、OptimizationException を捕捉できます。詳細については、「*エラー処理*」セクションを参照してください。

最適化のキャンセル

最適化の実行に長い時間かかる場合は、キャンセルする方法を提供することをお勧めします。Optimization API では簡単にキャンセルできます。最適化の実行中に Optimizer.CancelJob メソッドを呼び出すだけです。

最適化がすぐに停止しない場合もあります。最適化エンジンはできるだけ速やかに最適化をキャンセルし、JobDone イベントハンドラーを呼び出して、最適化が停止したことを知らせます。ハンドラーは、args 引数の Results プロパティにアクセスすることで、キャンセル前に処理された結果を取得できます。

通常、TradingApp は最適化をキャンセルするためのボタンを提供します。そのボタンの Click ハンドラーは CancelJob メソッドを呼び出します。アプリケーションに最適化を開始するためのボタンがある場合、同じボタンを使用して最適化をキャンセルすると便利です。最適化が開始されていない場合は、ボタンのテキストは「最適化を開始」(その他お好みのテキスト) にします。最適化が開始したとき、ボタンのテキストを「最適化をキャンセル」に変更します。最後に、最適化が終了したときに (正常またはキャンセルにより)、ボタンのテキストを「最適化を開始」に戻せます。

次のコードは、このパターンを実装する 1 つの方法を示すものです。

```
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;
```

上記の例は、最適化がキャンセルされた場合でも結果を処理することに注意してください。最適化が完了した場合のみ結果を処理する場合は、canceled が true のときに OptDone メソッドにより早く戻れます。

```
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;
```

エラー処理

Optimization API を扱う際に発生する可能性があるエラーには、3 つのカテゴリーがあります。

1. **ジョブ定義エラー。**ジョブを定義するコード内の回復不可能なエラーです。これらは常にコーディングエラーであり、ジョブをそれ以上定義できなくなるため、API は例外をスローしてエラーをレポートします。
2. **ジョブ検証エラー。**最適化の開始時に、ジョブ定義は完全性と正当性について検証されます。API は、`StartJob` メソッドから例外をスローすることで、検証エラーをレポートします。通常、これらもコーディングエラーですが、環境内の問題 (最適化に必要なストラテジーが現在の TradeStation 環境で定義されていないなど) によって発生する可能性もあります。
3. **実行時エラー**。最適化の開始後に発生するエラーです。API は、イベントを `JobFailed` ハンドラーに送信して、実行時エラーをレポートします。

次のセクションでは、これらのエラータイプのそれぞれを処理する方法について説明します。

ジョブ定義のエラーの捕捉

最適化ジョブを定義するコードは、ある種類のコーディングエラーに対して例外をスローします。最も一般的なケースでは、無効なインデックスをコレクションに渡します。この種類のエラーの後は続行する方法がないため、Optimization API は `OptimizationException` をスローします。

通常、これらのエラーは Event Log に表示されます。ただし、エラーの切り離しが困難な場合、エラーに関する詳細な情報をログに記録するために、ジョブ定義コードの各セクションを `try/catch` ブロックにラップできます。これらは致命的なエラーであるため、アプリケーションの実行を停止するために、例外を再スローすることをお勧めします。

たとえば、次のコードは例外をトリガーします。証券が追加される前に `Securities` オブジェクトへのインデックス付けを試行するためです。`catch` ブロックは、エラーに関するいくつかの情報を Print Log に出力します。その後、アプリケーションを停止するために例外を再スローします。

```
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;
```

これらのエラーは*常に*コーディングエラーであるため、リリース済みのアプリケーションでは、エラーをユーザーに直接レポートしないようにしてください。これらのエラーは主に、デバッギングを目的としたものです。

検証エラーの処理

ジョブ定義のエラーはたいてい、最適化の開始時に実行されるジョブ検証フェーズ中に捕捉されます。Optimization API は、Optimizer.StartJob メソッドから例外をスローすることで、これらのエラーをレポートします。次に、一般的な種類の検証エラーを示します。

- ジョブ定義が不完全です。たとえば、証券の足種を定義し忘れた場合、検証エラーとしてレポートされます。
- ジョブに無効なストラテジー名が含まれています。
- ジョブに無効な入力名が含まれています。

検証エラーのメッセージには、エラーを生成したノードまでのジョブ定義ツリー下の「パス」が含まれています。これにより、エラーの原因を特定しやすくなります。たとえば、ジョブ定義に無効な入力名が含まれているとします。生成されるエラーメッセージは、次のようになります。

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

StartJob メソッドは検証エラーに対して例外をスローするため、デフォルトで Event Log に表示されます。自分で検証エラーを処理する場合、try/catch ブロックに StartJob 呼び出しをラップします。たとえば、次のコードは Label コントロールの検証エラーを表示します。

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

「エラーパス」はデバッギングに非常に役立ちますが、ユーザーに検証エラーをレポートするときには省略できます(ほとんどの検証エラーは、正しいコーディングによって取り除くことができますが、環境内の問題によって発生する可能性もあります。たとえば、現在のワークスペースに、最適化に使用されるストラテジーが存在していない場合があります)。パスのないエラーメッセージを取得するには、error.Message ではなく error.MessageNoPath プロパティを使用します。

実行時エラーの処理

最適化が実際に実行するまで検出できないエラーは多数あります。実行時エラーが発生すると、Optimization API は最適化を停止して、JobFailed イベントを送信します。

実行時エラーを処理するには、JobFailed イベントハンドラーで次のステップを実行します。

1. args.Error プロパティからエラーオブジェクトを取得します。その後、エラーオブジェクトの Message プロパティを使用して、エラーメッセージを取得します。
2. エラーをユーザーにレポートします。

実行時エラーメッセージには「パス」が含まれていないため、Message プロパティと MessageNoPath プロパティは、そのエラーに対して同じ値を返します。

エラー処理の概要

Optimization API は、クライアントアプリケーションができる限りシンプルにエラー処理を実行できるようにするために、エラー処理を集中化するように設計されています。つまり必要なことは、次のケースの処理だけです。

- 実行時エラーを処理するために JobFailed イベントハンドラーを実装します。このハンドラーは、エラーメッセージをユーザーにレポートする必要があります。
- 検証エラーを Event Log に表示するのではなく、自分で処理する場合は、検証エラーを捕捉するために、try/catch ブロックの Optimizer.StartJob に呼び出しをラップします。
- ジョブ定義内のある種類のコーディングエラー (無効なインデックスエラーなど) をデバッグする必要がある場合、同様にこのコードも try/catch ブロックにラップできます。

最適化結果の取得

最適化が終了すると、アプリケーションは JobDone イベントハンドラー内で args.Results プロパティにアクセスすることで結果を取得します。これは、TradeStation の Strategy Optimization Report で使用可能なすべての情報を取得する tsopt.Results オブジェクトを返します。

上記の例は、Results クラスからいくつかのシンプルなメトリクスを取り出す方法を示しています。このセクションでは、その機能について詳しく説明します。

ストラテジーメトリクスと tsopt.MetricID

TradeStation での最適化は、Net Profit、Profit Factor、Percent Profitable など、ストラテジーのパフォーマンスを測定するさまざまな結果を計算します。これらはまとめて「適合度関数」と呼ばれることもあります。ただし、より正確に言うと、適合度関数は最適化しているパフォーマンスの *特定* の測定基準です。すなわち、Net Profit を最適化している場合、*それ* が適合度関数です。したがって、Optimization API では、特定の最適化の適合度関数として使用されるかどうかにかかわらず、ストラテジーパフォーマンスの測定基準を指す一般的な用語である「メトリック」が使用されます。

tsopt.MetricID 列挙は、使用可能な各メトリクスの識別子を提供します。これらの識別子の 1 つを Results.GetMetric メソッドに渡して、特定のメトリックのデータを取得できます。Results クラスも、すべてのストラテジーメトリクスに名前付きメソッドを提供します。GetMetric メソッドはより柔軟ですが、場合によっては、名前付きメソッドを使用するとコードがより読みやすくなります。

結果の構造

最適化結果はテストのセットとして格納されます。各テストには、次の情報が含まれます。

- テスト番号。テストがオプティマイザーにより評価された一連のテストに表示される場所を特定します。
- そのテストの各最適化パラメーターの値。通常、最適化パラメーターは入力値ですが、シンボル、

足種、またはストラテジーの有効ステータスであることもあります。

- そのテストの各ストラテジーメトリックのデータ。メトリックごとに、Results オブジェクトは、取引タイプ (AllTrades、LongTrades、または ShortTrades) と範囲タイプ (All、InSample、または OutSample) の各組み合わせの値を格納します。

さらに、Results オブジェクトはどのテストにも指定されず、全体として最適化を説明する値をいくつか提供します。

- テストの数。
- 最適化パラメーターの数。
- 各最適化パラメーターの見出し。最適化*入力値*の見出しには、Strategy Optimization Report と同様に、ストラテジーと入力した名前が含まれます。最適化*シンボル*の見出しは「DataN: Symbol」" (N はデータシリーズの数)、最適化*足種*の見出しは「DataN: Interval」です。
- 各ストラテジーメトリックの見出し。各見出しには、Strategy Optimization Report と同様に、取引タイプ (All、Long、または Short) とメトリック名が含まれます。

Results オブジェクトからデータを取得するとき、値のパラメーターインデックスやテストインデックスを渡す必要があることがあります。メソッドに両方が必要な場合は、必ずパラメーターインデックスを最初に渡します。これらのインデックスはゼロベースであることに注意してください。

Results オブジェクトが JobDoneEventArgs から取得されると、最適化に使用された適合度関数により、テストが最初にソートされます。ただし、Results クラスは、テスト番号だけでなく、任意のメトリックでテストを簡単にソートできるメソッドを提供します。

テストに関する情報の取得

Results オブジェクト内のテストの総数を取得するには、TestCount プロパティを使用します。通常、この数は最適化設定で指定した結果の数と同じになります (デフォルトで 200) が、パラメーターの組み合わせの数がその設定値未満の場合は、少なくなることがあります。テストの数は、結果の下部に同一の適合度を持つ複数のテストがある場合、指定した設定よりも多くなる可能性があります(Results オブジェクトには、必要な数よりもテストの数が多くなった場合でも、下部にすべての「適合関係」が含まれます。ただし、テストの数は、必要な数の 2 倍を超えることはありません)。したがって、Results オブジェクト内の実際のテストの数を決定するには、常に TestCount プロパティを使用してください。

特定のテストのテスト番号を取得するには、GetTestNum メソッドを呼び出し、テストのインデックスを渡します。

注: テスト*番号*とテスト*インデックス*を区別することが重要です。テスト*番号*は、テストがオプティマイザーにより評価された一連のテストすべてに表示される場所を特定します。テスト*インデックス*は、最終の一連の結果から取得するテストを特定するだけです。たとえば、results.GetTestNum(2) を呼び出して、3 番目のテストのテスト番号を取得できます。この場合、テスト*インデックス*が 2 であるのに対し、メソッドは、まったく異なる値になる可能性が高いテスト*番号*を返します。実際、テスト番号は TestCount よりも高くなります。これは、評価された *すべて*のテストの中からテストを特定するためで

す。ただし、テストインデックスは常に TestCount よりも低くなります。

TradeStation の Strategy Optimization Report を見ると、テスト*番号*はそのレポートの「Test」列と同じです。テスト*インデックス*は、1 ではなく 0 から始まることを除いて、行番号と同じです。結果のソート方法に応じて、同じテストインデックスが異なるテストを示す可能性がありますが、特定のテストのテスト番号は常に同じです。

ストラテジーメトリクスの取得

Optimization API は、Results オブジェクトからストラテジーメトリクスを取得するための 3 つの基本的なテクニックを提供します。

- 第 1 テストの特定メトリックを取得する名前付きメトリックメソッドを使用します。(テストを再ソートしていない場合、これは最良の適合度値を持つテストになります。)
- 特定のテストインデックス、取引タイプ、および範囲の特定メトリックを取得する名前付きメトリックメソッドを使用します。
- 特定の MetricID、テストインデックス、取引タイプ、および範囲のメトリックを取得する GetMetric メソッドを使用します。

次のセクションでは、これらの各テクニックの例を示します。

第 1 テストの名前付きメソッドの使用

引数なしで名前付きメトリックメソッドを使用すると、結果セット内の第 1 テスト、すべての取引タイプ、および履歴範囲全体のそのメトリックを取得します。前述のように、適合度により、最初にテストがソートされます。したがって、テストを再ソートしていない場合、引数なしでテストを呼び出すと、「最良」のテストの値が取得されます。-

たとえば、Net Profit を最適化の適合度関数として使用するとします(これがデフォルトです)。テストを再ソートしていないと仮定すると、次のコードにより、最良の Net Profit を有するテストの Profit Factor が取得されます。-

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

特定のテスト、取引タイプ、範囲の名前付きメソッドの使用

名前付きメトリックメソッドを呼び出して、テストインデックス、取引タイプ (AllTrades、LongTrades、または ShortTrades)、および結果範囲 (All、InSample、および OutSample) を渡すこともできます。たとえば、次のコードは、ロング取引およびインサンプル結果の第 3 テストの Percent Profitable を取得します。

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

GetMetric Method の使用

GetMetric メソッドを呼び出して、任意のテスト、取引タイプ、および結果範囲に使用可能なメトリック

を取得できます。最初の引数は、取得対象のメトリックを指定する tsopt.MetricID 値です。残りの引数は、テストインデックス、取引タイプ、および結果範囲です。

たとえば、次のコードは、ショート取引およびアウトオブサンプル結果の第 5 テストの Gross Profit を取得します。

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

GetMetric メソッドは、名前付きメトリックメソッドよりも冗長ですが、より柔軟です。目的のメトリックが引数として渡されるため、MetricID のセットをループし、それをメソッドに渡して、各メトリックの値を取得できます。このセクションの後半にある拡張した例には、このテクニックの例が含まれています。

最適化されたパラメーターデータの取得

ストラテジーメトリクスは、テスト結果の重要な要素です。その他の重要な要素は、そのメトリクスを生成した最適化されたパラメータ値です。

たとえば、アプリケーションが「MovAvg Cross」ストラテジーへの FastLen 入力と SlowLen 入力を最適化するとします。特定のテストの Net Profit を有意にするため、そのテストに使用される FastLen と SlowLen の値を知る必要があります。

Results クラスの次のメソッドは、最適化パラメーターに関する情報を提供します。

- OptParamCount プロパティは、最適化パラメーターの数を返します。
- GetOptParamHeading メソッドは、特定の最適化パラメーターの記述的な見出しを返します。これは特定のテストに固有ではないため、メソッドはパラメーターインデックスのみを受け付けます。移動平均の例では、最初のパラメーター見出しは「MovAvg Cross: FastLen」になり、2 番目のパラメーター見出しは「MovAvg Cross: SlowLen」になります。
- GetOptValueString メソッドは、特定の最適化パラメーターとテストの値の文字列表現を返します。このメソッドには、パラメーターインデックスとテストインデックスの 2 つの引数が必要です。文字列が返されますが、これはシンボル、足種、式入力など、最適化パラメーターを表すのに十分な柔軟性があるためです。移動平均の例では、メソッドは「7」や「15」などの値を返します。最適化シンボルの場合、「CSCO」や「MSFT」などの値を返します。最適化足種の場合、「5 min」や「10 min」などの値を返します。

このセクションの後半の拡張した例では、これらのメソッドの使用方法を示します。

特定の種類のパラメーターデータの取得

GetOptParamHeading メソッドと GetOptParamString メソッドは、ジョブのすべての最適化パラメーターに関する一般情報を取得するのに便利です。ただし、特定の種類のテスト用パラメーターを取得する場合があります。たとえば、特定のテストに使用するシンボルまたは特定の入力の値を知る必要がある場合があります。すべてのパラメーター見出しを反復して文字列を解析し、表すデータのタイプ (シンボル、足種、入力値など) を決定することができますが、もっと簡単な方法があります。

tsopt.Results クラスは、特定の種類の特定テスト用パラメーターデータを取得する次のメソッドを提供します。

- GetTestSymbol メソッドは、テストに使用されるシンボルを返します。最初の引数はテストインデックスです。マルチデータストラテジーの場合、シンボルのデータシリーズを示す省略可能な 2 番目の引数を渡すこともできます (Data1 は 1、Data2 は 2 など)。2 番目の引数を省略した場合、メソッドは Data1 のシンボルを返します。
- GetTestInterval メソッドは、テストに使用される足種を返します。GetTestSymbol と同様に、最初の引数はテストインデックスで、省略可能な 2 番目の引数はデータシリーズです。このメソッドは、文字列ではなく、tsopt.Interval オブジェクトを返すことに注意してください。足種の文字列表現を取得する場合は、返される Interval オブジェクトで Describe メソッドを呼び出します。Interval オブジェクトで別のメソッドまたはプロパティを呼び出して、足種の詳細情報を取得することもできます。
- GetTestInputString メソッドは、テストに使用される特定の入力値の文字列表現を返します。テストインデックス、入力名、ストラテジー名を最初の 3 つの引数として渡す必要があります。ジョブ定義が同じストラテジーのインスタンスを複数含んでいる場合、どのインスタンスを使用するかを示す 4 番目の引数も渡す必要があります (そのストラテジーの最初のインスタンスは 0、2 番目のインスタンスは 1 など)。このメソッドは値を文字列として返すので、テキスト入力、式入力、Boolean 入力など、あらゆる種類の入力を処理する十分な柔軟性があります。
- GetTestInputDouble メソッドは GetTestInputString と同様に機能しますが、入力値を double 型として返して、文字列を数値に変換する手間を省きます。このメソッドは、指定した入力が数値であることがわかっている場合にのみ使用してください。それ以外の場合、メソッドは例外をスローして、入力値を double 型に変換できないことを示します。
- IsTestStrategyEnabled メソッドは、特定のストラテジーがテストに対して有効化されているかどうかを示します。これは、OptStrategyEnabled メソッドを使用して、ストラテジーの有効ステータスを最適化し、そのストラテジーが特定のテストに対して有効または無効にされたかどうかを知りたい場合に役立ちます。テストインデックスとストラテジー名を最初の 2 つの引数として渡す必要があります。ジョブ定義が同じストラテジーのインスタンスを複数含んでいる場合、どのインスタンスを使用するかを示す 3 番目の引数も渡す必要があります (最初のインスタンスは 0、2 番目のインスタンスは 1 など)。

次に、これらのメソッドの例を示します。

```
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");
```

これらのメソッドには、最適化パラメーターと非最適化パラメーターの両方で有効であるという重要な特

長があります。たとえば、上記の GetTestInputDouble の例は、「Length」入力が最適化されるかどうかに関係なく、その入力値を取得します。入力が最適化される場合、戻り値はテストごとに異なります。入力が最適化されない場合、その入力の戻り値は、すべてのテストで同じ値になります。

結果のソート

Results クラスは、テストをソートする次の 2 つのメソッドを提供します。

- SortByTestNum メソッドは、テスト番号でテストをソートします。
- SortByMetric メソッドは、特定のメトリック、取引タイプ、結果範囲でテストをソートします。

どちらのメソッドも、昇順または降順でソートできます。降順でソートするには、reverse 引数に true を渡します。

たとえば次のコードは、テスト番号で昇順に結果をソートします。

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

次のコードは、Profit Factor で降順 (最大値から最小値) に結果をソートします。すべての取引 (ロングおよびショート) のインサンプル結果を使用します。

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

例: テキストファイルへの結果の書き込み

次の例では、Results クラスのさまざまなメインの機能について説明します。特定の取引タイプと結果範囲の最適化結果をカンマ区切りのテキストファイルに書き込みます。ファイルの形式は、TradeStation の Strategy Optimization Report と非常によく似ています。

重要: tsopt.Results クラスでは、以下のコードと同じファイルを生成できる WriteFile メソッドをすでに提供しています。それでも以下のコードは、tsopt.Results クラスの使い方の例として、さらに、独自にカスタマイズしたバージョンの最適化結果を保存する場合のテンプレートとして有益です。Results.WriteFile メソッドについては、例のあとで説明します。

このメソッドは、最適化の結果を保存するために、JobDone イベントハンドラーから呼び出されます。

```
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;
```

例では GetMetric メソッドを使用して、メトリクスデータを取得していることに注意してください。このメソッドは目的のメトリックを特定する tsopt.MetricID を受け付けるため、コードはすべてのメトリクスで反復し、その結果を書き出します。これは、メトリックごとに名前付きメソッド (NetProfit、GrossProfit など) を呼び出すよりも簡潔です。

このコードは、静的な Results.GetMetricHeading メソッドも呼び出して、各 MetricID の見出しも取得します。このメソッドは、Strategy Optimization Report の見出しと同様に、取引タイプを各メトリック名の前に付けます。取引タイプの接頭辞を付けずにメトリック名を取得するには、代わりに Results.GetMetricName メソッドを呼び出します。

GetOptValueString メソッドが、カンマまたは引用符を含む文字列を返す場合もあります (式やテキスト値を最適化している場合など)。したがって、このような場合に有効な CSV フィールドを返すヘルパーメソッドも提供します。このメソッドにより、カンマや引用符を含むフィールドが確実に引用符で囲まれ、埋め込まれた引用符が確実にエスケープされます (2 つの引用符に変換)。
GetOptValueString の戻り値を CSV_Field に渡し、変換した文字列をファイルに書き込みます。これにより、Microsoft Excel® などのツールでファイルを正しく解析し開くことができるようになります。

簡単な方法での結果の書き込み

Strategy Optimization Report と同じ基本的な形式のファイルに結果を書き込む場合、それを行う独自のコードを作成する必要はありません。tsopt.Results クラスには、代わりにすべての作業を実行する WriteFile メソッドが含まれています。

Results.WriteFile メソッドには次のシグニチャーがあります。

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

カンマを区切り記号として渡した場合、このメソッドは有効な CSV ファイルを生成します。

次に、このメソッドを使用して結果を書き込むシンプルな JobDone イベントハンドラーを示します。

```
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;
```

キューに置かれる最適化

アプリケーションの多くは、一度に 1 つの最適化のみを実行する必要があります。ただし、複数の最適化を設定して、すべて順々に実行する場合があります。Optimization API では、これをとても簡単に行うことができます。

job1 および job2 という変数に格納される 2 つの最適化ジョブを定義すると仮定します。ジョブごとに StartJob メソッドを呼び出すと、順々に 2 つのジョブを自動的に実行できます。

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

これにより、次の一連のイベントが開始します。

1. オプティマイザーが job1 の最適化を直ちに開始し (使用可能なコアをすべて使用)、job2 をそのジョブキューに配置します。
2. job1 の最適化が終了すると、結果を処理できるように、オプティマイザーが JobDone イベントハンドラーを呼び出します。次に、job2 の最適化を開始します。
3. job2 の最適化が終了すると、2 つ目のジョブの結果を処理できるように、オプティマイザーが JobDone イベントハンドラーを再び呼び出します。

ジョブ ID

上記の例では、オプティマイザーが JobDone イベントハンドラーを 2 回呼び出すため、どのジョブが終了したかを知る方法が必要になります。

その方法は、イベントハンドラー内の args 引数の JobID プロパティにより提供されます。これは、アプリケーション内でキューに配置された各ジョブを一意に識別する 64 ビットの整数です。このプロパティにアクセスすると、どのジョブが終了したかを判断できます。

JobID は最適化ジョブの開始時に割り当てられ、StartJob メソッドにより返されます。複数のジョブをキューに入れていない場合、前の例で行ったように、戻り値を無視しても問題ありません。ただし、キュー機能を使用しているとき、StartJob により返された値を変数に割り当てる場合があります。たとえば次のコードは、IDJob1 および IDJob2 という変数に ID を格納します。その後、イベントハンドラーは JobID プロパティを各変数と比較して、どのジョブが完了したかを決定します。

```
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;
```

JobID を格納する一般的なメソッドとしては、Dictionary オブジェクトを使用して、各 ID を名前に関連付ける方法があります。次の例は、2 つのディクショナリーを使用します。1 つは各名前をジョブ定義に関連付け、もう 1 つは各 JobID を名前に関連付けます。これにより、JobID の名前を取得し、その名前のジョブ定義を取得するのが容易になります (必要な場合)。

```
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;
```

これで、JobDone イベントハンドラーは、これらのディクショナリーを使用して、完了したジョブに関する有意な情報を取得できるようになりました。たとえば、次のイベントハンドラーはジョブ名を取得し、それを使用してファイル名を作成します。その後、そのファイルに最適化結果を書き込みます。

```
// 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;
```

この例が示すように、JobID ごとに異なるコードを常に作成する必要はありません。同じ方法ですべての最適化の結果を処理する場合は、JobDone イベントハンドラーに 1 度ロジックを作成するだけで、すべてのジョブを処理できます。

JobDone イベントハンドラーに焦点を当ててますが、JobFailed イベントハンドラーと ProgressChanged イベントハンドラーの args オブジェクトにも JobID プロパティがあります。このプロパティを使用して、どのジョブがイベントをトリガーしたかを判断できます。

キューに置かれた最適化のキャンセル

複数の最適化をキューに入れているとき、次の 3 つの方法でジョブをキャンセルできます。

- 引数なしで CancelJob メソッドを呼び出します。オプティマイザーは現在実行しているジョブをキャンセルし、そのジョブの JobDone イベントハンドラーを呼び出します。キュー内にジョブが残っている場合、オプティマイザーは次のジョブを開始します。
- CancelJob メソッドを呼び出し、キャンセルするジョブの JobID を渡します。指定したジョブが現在実行中の場合、オプティマイザーはそれをキャンセルして、JobDone イベントハンドラーを呼び出し、次のジョブを開始します (ある場合)。それ以外の場合、オプティマイザーは指定したジョブをキューから削除します。
- CancelAllJobs メソッドを呼び出します。オプティマイザーは現在実行しているジョブをキャンセルし、そのジョブの JobDone イベントハンドラーを呼び出します。キュー内にジョブが残っている場合、オプティマイザーはそのジョブを削除します。

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 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 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 call after the last property, and this will make the definition a valid statement.

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

Job Class

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

`static tsopt.Job Create();`

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

`Object Clone();`

Creates a copy of the job definition. Since is a standard method of the class, it returns the job as an . In order to use the cloned job, you will need to cast it to 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.Job SetOptimizationMethod(tsopt.OptimizationMethod optMethod);

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

property tsopt.Securities Securities { read; }

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

property tsopt.Settings Settings { read; }

Returns a object which represents the settings to apply to the entire optimization. Note that settings for a specific strategy are accessed using a object rather than the 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 object with the file name.

Securities Class

The 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 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 object at the specified index. The object represents a collection of 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 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 argument is false, the method will not modify the job definition, and it will return false if any symbols are invalid.

If the 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 , but this version also adds any invalid symbols to the vector. You must create the object and pass it to the method.

Security Class

The 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 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 rather than via or).

```
property bool IsIntervalOptimized { read; }
```

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

```
property string Symbol { read; write; }
```

Gets or sets the symbol for the security.

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

tsopt.Security SetSymbol(string symbol);

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

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

property tsopt.Interval Interval { read; }

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

If the interval is currently optimized for a security, modifying the returned 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 instead.

property tsopt.History History { read; }

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

property tsopt.OptSymbol OptSymbol { read; }

Gets an object that represents an optimized symbol for the security. You can call methods on the 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 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 instead.

property tsopt.OptInterval OptInterval { read; }

Gets an object that represents an optimized interval for the security. You can call methods on the 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 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 instead.

property tsopt.SecurityOptions SecurityOptions { read; }

Gets the 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 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 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 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 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 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 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 and . The and arguments determine how the bars are built.

The argument can be either or .

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

Sets a Kase interval based on the underlying time interval specified by . The resolution quantity for Kase bars is always 1. The 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 and . The 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 and . The 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 and . The other arguments determine how the bars are built.

The argument can be either or .

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

Sets a Range Chart interval based on the underlying time interval specified by and . The 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 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 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 and 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 of and a 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 would return and 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 is ; otherwise, its value is not meaningful.

The value returned by this property can be either or .

property double KagiReversalSize { read; }

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

The value returned by this property can be either or .

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 is ; 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 is ; 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 is ; 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 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 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 property.

If is , the 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 is ; 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 , the API will throw an 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 , the API will throw an 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 is ; 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 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 value. This method also returns the 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 object, so it can be used in tree-style definitions.

If the string has an unrecognized date format, the API will throw an 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 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 value. This method also returns the 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 object, so it can be used in tree-style definitions.

If the string has an unrecognized date format, the API will throw an 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 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 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 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 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 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 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 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 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 . Use the 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 object, so you can chain 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 object, so you can chain 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 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 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 object at the specified index. The 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 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 . Use one of the methods to add intervals to the list.

tsopt.Interval AddInterval();

Adds a default 5 minute interval to the list and returns the 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 methods in the class return the object so that you can call additional methods on that object. This allows 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 and . The and arguments determine how the bars are built.

The argument can be either or .

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

Adds a Kase interval based on the underlying time interval specified by . The resolution quantity for Kase bars is always 1. The 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 and . The 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 and . The 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 and . The other arguments determine how the bars are built.

The argument can be either or .

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

Adds a Range Chart interval based on the underlying time interval specified by and . The 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 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 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 and 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 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 property of a object.

```
property tsopt.Security EndSecurityOptions { read; }
```

Returns the parent object in the job definition tree, which is a 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 object so that you can chain the methods together. This is useful when specifying options within a tree-style job definition.

Strategies Class

The 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 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 object at the specified index. The object acts as a collection of 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 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 object. If 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 object to get the inputs and display them to the user.

Note: Populating the 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 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 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 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 object that defines the inputs for the strategy. This includes both fixed inputs and optimized inputs. You can call methods on the 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 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 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 class in this *API Reference* for more information.

property tsopt.IntrabarOrders IntrabarOrders { read; }

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

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

ELInputs Class

The 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 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 property does not necessarily return the total number of inputs for the strategy as it is defined in EasyLanguage. Instead, the 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, may return a number lower than the total input count for the strategy.

If you pass true for the 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 property will return the total input count for the strategy.

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

Returns the object for the specified index. The object acts as a collection of 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 object is to find out the properties of the input. To set or optimize an input, you must use other methods from the 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 . If you pass true for the argument when you add a strategy, it will automatically add all of

the strategy's inputs to the definition. Otherwise, the object will contain only those inputs that you explicitly define with either the method or one of the methods in the 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 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 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 that accepts the value as a double (see below).

If the 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 argument indicates the type of the input. Use this version of if the input has a non-numeric type.

The 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 argument of 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 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 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 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), 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 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 value and incrementing the value by until it reaches the value.

Note that this method merely specifies that the input should be optimized. The actual optimization occurs when you call the 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 object that represents the list of values. You can call the method on that object to add values to the list.

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

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

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

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

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

ELInput Class

The 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 class.

An object can be obtained by indexing into an 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 property returns an empty string. You can use the 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 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 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 property returns . Otherwise, accessing this property will throw an .

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 of type .)

```
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 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 property returns . Otherwise, accessing this property will throw an .

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 of type .)

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

The input values for an follow the same rules as a single input value. See the method in the 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 class provides the following methods and properties.

property tsopt.ELInputs EndOptList { read; }

Returns the parent object in the job definition tree, which is an 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 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 . You can use one of the 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 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 object in a local variable. Then you can call the method on that variable within the loop. For example, the following code optimizes the "BollingerPrice" input using the values in a .

```
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 method also returns the object, so you can chain 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 class allows you to get or set individual signal states for a strategy. You can obtain a object by accessing the property on a 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 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 class provides the following methods and properties.

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

Returns the parent object in the job definition tree, which is a 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 class allows you to get or set the intrabar order generation options for a strategy. You can obtain an object by accessing the property on a object.

The class provides the following methods and properties.

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

Returns the parent object in the job definition tree, which is a 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 option is not truly unlimited for intrabar entries. The number of intrabar entries will not exceed the value specified by the 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 is .

```
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 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 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 object from the property on a 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 class provides the following methods and properties.

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

Returns the parent object in the job definition tree, which is a 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 property in the class).

This class has the following members:

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

Returns the parent object in the job definition tree, which is a 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 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 object so that you can chain the methods together. This is useful when specifying options within a tree-style job definition.

ResultOptions Class

The 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 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 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 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 () or the lowest fitness values ().

```
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 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 object so that you can chain the methods together. This is useful when specifying options within a tree-style job definition.

GeneralOptions Class

The 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 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 object so that you can chain the methods together. This is useful when specifying options within a tree-style job definition.

CostsAndCapital Class

The 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 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: , , , .

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: , , or .

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 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 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 object so that you can chain the methods together. This is useful when specifying options within a tree-style job definition.

PositionOptions Class

The 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 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 object so that you can chain the methods together. This is useful when specifying options within a tree-style job definition.

TradeSize Class

The 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 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 .

```
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 .

```
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 .

```
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 .

```
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 object so that you can chain the methods together. This is useful when specifying options within a tree-style job definition.

BackTesting Class

The 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 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 shares have traded at limit price or better.)
- tsopt.FillModel.TradeBased (Fill order at limit price or better after 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 is .

```
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 is .

```
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 object so that you can chain the methods together. This is useful when specifying options within a tree-style job definition.

OutSample Class

The 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 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 , the property will throw an exception.

If you *set* this property value and the sample type is not , the sample type will automatically be changed to and the 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 , the property will throw an exception.

If you *set* this property value and the sample type is not , the sample type will automatically be changed to and the 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 , the property will throw an exception.

If you *set* this property value and the sample type is not , the sample type will automatically be changed to and the 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 , the property will throw an exception.

If you *set* this property value and the sample type is not , the sample type will automatically be changed to and the 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 object so that you can chain the 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 .

To use the entire range of bars as the in-sample, pass zero for both and . 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 or following will constitute the out-of-sample range.

WalkForward Class

The 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 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 object so that you can chain the methods together. This is useful when specifying options within a tree-style job definition.

CommissionBySymbol Class

The 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 object from the property of a object.

The commissions for each symbol use the that you specify in the object. If you don't explicitly set the commission mode, the default is .

If a symbol used in the optimization is not found in the object, the optimizer will use the specified in the 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 method below for more information.

property tsopt.CostsAndCapital EndCommissionBySymbol { read; }

Returns the parent object in the job definition tree, which is a 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 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 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 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 object, so you can chain together multiple 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 object, so you can chain together multiple calls.

tsopt.CommissionBySymbol Clear();

Clears all the symbol entries from the object.

void ReadFile(string fileName);

Reads the symbol entries from a text file. The 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 method to write the contents of a object that has been populated by your TradingApp.

void WriteFile(string fileName);

Writes the symbol entries in a object to a text file. The argument should specify the full path to the file.

This method writes the data using the format described under the method above.

SlippageBySymbol Class

The 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 object from the property of a object.

The slippage values for each symbol use the that you specify in the object. If you don't explicitly set the slippage mode, the default is .

If a symbol used in the optimization is not found in the object, the optimizer will use the specified in the 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 method below for more information.

property tsopt.CostsAndCapital EndSlippageBySymbol { read; }

Returns the parent object in the job definition tree, which is a 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 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 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 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 object, so you can chain together multiple 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 object, so you can chain together multiple calls.

tsopt.SlippageBySymbol Clear();

Clears all the symbol entries from the object.

void ReadFile(string fileName);

Reads the symbol entries from a text file. The 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 method to write the contents of a object that has been populated by your TradingApp.

`void WriteFile(string fileName);`

Writes the symbol entries in a object to a text file. The argument should specify the full path to the file.

This method writes the data using the format described under the method above.

AvailableStrategies Helper Class

The 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 class provides a simple way to do that.

To use the class, just create an object of the class and iterate through it like a collection. You can use the 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 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 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 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 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 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 property.) Thus, the class does not return the name of the default session, since it is always an empty string.

To use the class, just create an object of the class and iterate through it like a collection. You can use the 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 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 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 property instead. This substitution is required for the default session, since it is always represented by an empty string in the Optimization API.

The 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 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 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 object:

```
optimizer = tsopt.Optimizer.Create();
```

You may also use the keyword to create the object:

```
optimizer = new tsopt.Optimizer;
```

`JobDone Event`

The optimizer calls your 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 event handler is called instead.

Before starting an optimization, you must assign an event handler to the 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 object, be sure to assign the event handler to the event:

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

The event handler is where you retrieve and process the results of an optimization. The results are available as a property of the argument.

If your event handler calls many methods and/or properties on the 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 property of the argument. If the job was canceled, the object will contain any results that were processed

before the cancellation.

JobFailed Event

The optimizer calls your event handler when an error occurs during an optimization.

Before starting an optimization, you must assign an event handler to the 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 object, be sure to assign the event handler to the event:

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

A event handler should report the optimization error to the user. The error object can be retrieved from the property of the argument. You can get the error message by accessing the 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 event handler periodically during an optimization in order to provide information about the progress of the optimization.

The 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 and properties of the argument. For more information about the objects returned by these properties, see the entries below for the and classes.

The following 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 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 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 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 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 . 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 call inside a 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 method. If the value of 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 event handler. The results for all completed optimization tests will be available in the object.

void CancelJob(int64 jobID);

Cancels the optimization with the specified (which is returned by the method). This version of the 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 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 event handler. Any waiting jobs will simply be removed from the queue.

JobDoneEventArgs Class

A object is passed as the second argument of your 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 call that started the optimization.

The 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 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 class, see its entry below in the *API Reference*.

property bool Canceled { read; }

Returns true if the event handler was called because the optimization job was canceled.

Note that when an optimization is canceled, the object will contain any results that were processed before the cancellation.

JobFailedEventArgs Class

A object is passed as the second argument of your event handler. It provides information about the optimization job that just failed.

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 call that started the optimization.

The 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 object that provides error information about the job that just failed. You can access the property of this object to get the error message.

ProgressChangedEventArgs Class

A object is passed as the second argument of your 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 call that started the optimization.

The 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 object that contains information about the progress of the running optimization, such as the current test number.

For complete information about the class, see its entry below in the *API Reference*.

property tsopt.BestValues BestValues { read; }

Returns a object that contains information about the best fitness result seen so far in the optimization.

For complete information about the class, see its entry below in the API.

ProgressInfo Class

The class provides progress information about the optimization that is current running.

You can get a object by accessing the property of the object, which is passed as the second argument of your 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 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 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 object by accessing the property of the object, which is passed as the second argument of your 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 argument is the zero-based index of the parameter. You can loop from to 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 argument is the zero-based index of the parameter. You can loop from to 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 class:

- If there is a non-recoverable error in your job definition code, the API will throw an from that code.
- If there is a validation error in your job definition, the API will throw an from the method.
- If an error occurs during an optimization, the optimizer will call your event handler. You can retrieve the error from the property, which returns an object.

The class inherits from the 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 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 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 and properties will always return the same message within a event handler.

property tsopt.ErrorCode ErrorCode { read; }

This is a special property provided by the class. It returns a value from the 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 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 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 enumeration, search for "ErrorCode" in the Dictionary window of the TradeStation Development Environment; then click on the entry.

The Results Class

The class provides complete information about the results of an optimization.

When the optimizer calls your event handler, it passes a object as the second argument (). You can access the property of this object to get the object for the optimization.

For a detailed discussion about using the class, see *Retrieving Optimization Results* earlier in this guide.

`property int TestCount { read; }`

Returns the number of optimization tests in the object.

Note that you can specify the number of tests to keep by setting the property in the object, which is a sub-object of . (The default value of is 200.) However, the value returned by will not necessarily be the same as this number. It may be *less* if the number of parameter combinations is lower than . The count can also be *higher* if there are multiple tests with identical fitness at the bottom of the results. (The 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 property to determine the actual number of tests in the 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 object: 0 indicates the first test, 1 indicates the second test, and so on. This number must always be less than .

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 , 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 object. If you want a string representation of the interval, you can call the method on the object. You may also call other methods or properties on the 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 argument must be the name of one of the strategies defined in the optimization job. Otherwise, the method will throw an with a message that the strategy was not found.

The argument must be the name of one of the inputs in that strategy. Otherwise, the method will throw an 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 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 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 , 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 , 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 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 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 argument is the zero-based index of the optimized parameter. It should always be less than .

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 argument is the zero-based index of the optimized parameter. It should always be less than .

The index argument is the zero-based index of the test within the results. It should always be less than .

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 .

This is a static method, so you should call it as .

For example, the following code will assign “Net Profit” to the variable:

```
name = tsopt.Results.GetMetricName(tsopt.MetricID.NetProfit);
```

static string GetMetricHeading(tsopt.MetricID metricID, tsopt.TradeType type);

Returns the metric heading for the specified and . 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 .

For example, the following code will assign "Long: Net Profit" to the 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 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 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 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 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 argument indicates whether to sort the tests in reverse order (highest to lowest).

The test number is the value returned by the 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 argument should be the job definition that produced the optimization results. This is important because 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 or when you try to run the modified job.

The 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 in our 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 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 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, replaces the optimized inputs (which were specified by) with the fixed input values from the test.

The Optimization API allows you to optimize other kinds of parameters besides inputs, and works equally well for these. For example, if a job optimizes over symbols, then the method will replace the optimized symbol in the job definition with the fixed symbol from the specified test in the results.

The 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 argument should specify the full path to the file you want to write.

The 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 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 and 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:

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).

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 function and passing the relevant 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 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 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:

The call returns a reference to the object, so that we can call another method on the job.

The property returns a object, which provides methods to add, get, or delete Security nodes.

The method adds a security and returns a object that represents that node. This object provides methods to define a security.

The method sets the symbol and returns a reference to the object so that we can call another method or property on the security.

The property returns an object, which provides methods to set different kinds of intervals. We call the method to define the interval.

The property returns the parent object of the , which is a object.

We can now access , which is another property of the object. This provides various methods to set the history range. Here we call the and methods.

Then we can call to return to the object again.

We've now defined all the key parts of a security (symbol, interval, and history), so we call to get back to the object.

We don't need to add a second security for this job, so we call to get back to the object.

Since we are in the context of the object again, we can now build up a strategies sub-tree using the same basic techniques. We start by using the property to get a object.

The method adds a strategy with the specified name and returns a object, which provides properties and methods to define information about the strategy.

The property returns an object for the strategy. This object provides various methods for either setting or optimizing inputs. Here we call the method twice to optimize two different inputs. Note that the method returns the object so that we can call additional methods on it.

The property returns the parent object of the object, which is a 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 property, which returns the parent object.

We repeat this process to add a second strategy and optimize two of its inputs.

After adding the two strategy definitions, we use the property to get back to the object.

Now we want to change a few default settings, so we use the property to get the object.

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 property to get a object. Then we call a couple methods to set some options. Each of these methods returns the object so that we can set additional options. Finally, we use the property to get back to the object.

We repeat this process to change a couple of the result options. Then we call to get back to the object and to return to the object.

At the end of the definition, we must call the method to complete the definition. The 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 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 method for the property. Every writable property has a corresponding 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 and methods both return the 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 property in the example above returns the sub-object of the object, and the property returns the sub-object of the 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 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 property.
- Some methods also return sub-objects. For example, the method creates and returns a sub-object, and the method creates and returns a 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 property. If the last statement in your definition is an property, add a call to complete the definition. On the other hand, if you have written a partial job definition that just chains some calls together, you can omit both the property and the 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.
- properties that return the parent object. Press Shift+Tab to return to the previous level before typing the property.
- 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.
- methods that add a value or string (e.g.). These just return the method’s object so that you can chain the calls together. Don’t change the indentation until the next property.
- 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 property.
- 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 method requires additional information (the values to optimize), so it returns an sub-object.

Note: and are defined as *properties* of the 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 , 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 or 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.