TRADING
TECHNOLOGIES

WRITING MULTI-THREADED APPLICATIONS WITH TT API 7.X

VERSION 7.X

DOCUMENT VERSION 7.X.DV1 3/5/14

LEGAL

1 Threading and TT API

Introduction

About Dispatchers

The TT API is based on an event-driven model. Applications can subscribe for instrument, price, order, and fill events. When an event occurs, the TT API posts a message to a queue managed by a Dispatcher. The Dispatcher then delivers the messages to the observers that have subscribed for the events. Therefore, each thread that interacts with TT API must have a Dispatcher. You can associate only one Dispatcher with each thread.

If you want to access TT API functionality from a thread on which you created either Windows Forms or WPF Controls, you can direct the TT API to use the Windows GUI event queue and Dispatcher for the current thread as follows:

```
UIDispatcher m_disp = Dispatcher.AttachUIDispatcher();
```

If you do not create Windows Forms or WPF Controls on a thread, you need to direct TT API to create an event queue and a TT API Dispatcher (`TradingTechnologies.TTAPI.Dispatcher`) for the current thread and start it as follows:

```
WorkerDispatcher m_disp = Dispatcher.AttachWorkerDispatcher();
m_disp.Run();
```

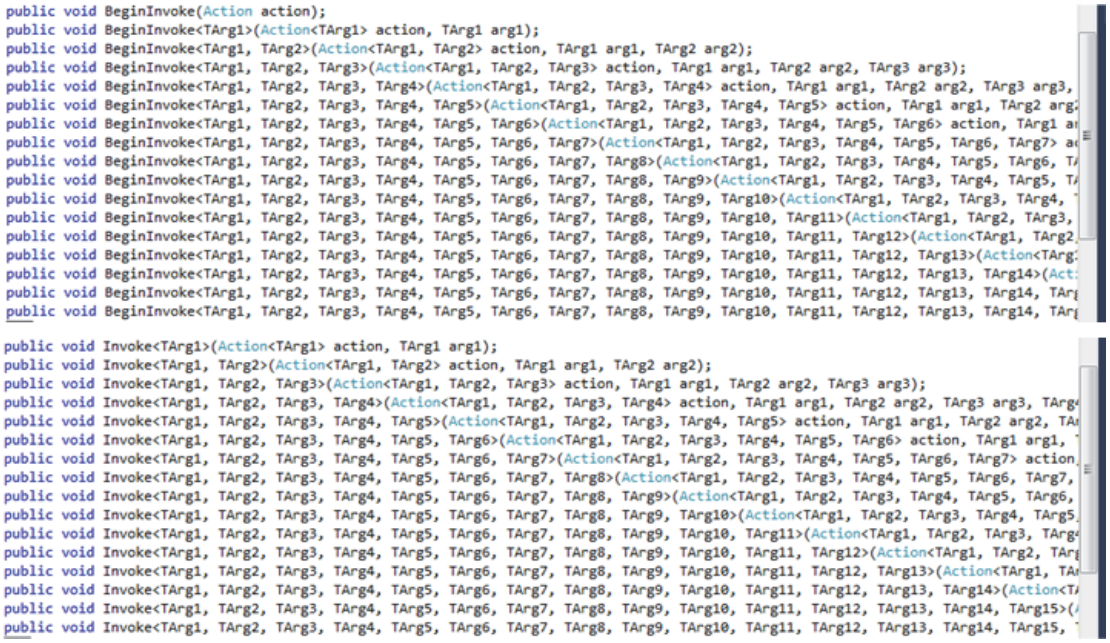
Calling either of these static methods also stores a reference to the Dispatcher in a publicly accessible static property named **`Dispatcher.Current`**.

Normally, a thread exits when it completes its work. However, threads with associated Dispatchers keep a thread alive until either the Dispatcher is stopped or the thread is terminated. To stop the TT API Dispatcher, you need to call either the **`InvokeShutdown`** or the **`BeginInvokeShutdown`** method as follows.

```
// Shutdown the Dispatcher
if (m_disp != null)
{
  m_disp.BeginInvokeShutdown();
}
```

Posting Messages to the Dispatcher

When using the **`Dispatcher.AttachWorkerDispatcher`** method, you must use either the **`Invoke`** or **`BeginInvoke`** methods defined in the `TradingTechnologies.TTAPI.Dispatcher` class to post messages to the event queue. These methods have many different signatures, as shown in the following illustration.



```
public void BeginInvoke(Action action);
public void BeginInvoke<TArg1>(Action<TArg1> action, TArg1 arg1);
public void BeginInvoke<TArg1, TArg2>(Action<TArg1, TArg2> action, TArg1 arg1, TArg2 arg2);
public void BeginInvoke<TArg1, TArg2, TArg3>(Action<TArg1, TArg2, TArg3> action, TArg1 arg1, TArg2 arg2, TArg3 arg3);
public void BeginInvoke<TArg1, TArg2, TArg3, TArg4>(Action<TArg1, TArg2, TArg3, TArg4> action, TArg1 arg1, TArg2 arg2, TArg3 arg3,
public void BeginInvoke<TArg1, TArg2, TArg3, TArg4, TArg5>(Action<TArg1, TArg2, TArg3, TArg4, TArg5> action, TArg1 arg1, TArg2 arg
public void BeginInvoke<TArg1, TArg2, TArg3, TArg4, TArg5, TArg6>(Action<TArg1, TArg2, TArg3, TArg4, TArg5, TArg6> action, TArg1 a
public void BeginInvoke<TArg1, TArg2, TArg3, TArg4, TArg5, TArg6, TArg7>(Action<TArg1, TArg2, TArg3, TArg4, TArg5, TArg6, TArg7> a
public void BeginInvoke<TArg1, TArg2, TArg3, TArg4, TArg5, TArg6, TArg7, TArg8>(Action<TArg1, TArg2, TArg3, TArg4, TArg5, TArg6, TA
public void BeginInvoke<TArg1, TArg2, TArg3, TArg4, TArg5, TArg6, TArg7, TArg8, TArg9>(Action<TArg1, TArg2, TArg3, TArg4, TArg5, TA
public void BeginInvoke<TArg1, TArg2, TArg3, TArg4, TArg5, TArg6, TArg7, TArg8, TArg9, TArg10>(Action<TArg1, TArg2, TArg3, TArg4,
public void BeginInvoke<TArg1, TArg2, TArg3, TArg4, TArg5, TArg6, TArg7, TArg8, TArg9, TArg10, TArg11>(Action<TArg1, TArg2, TArg3,
public void BeginInvoke<TArg1, TArg2, TArg3, TArg4, TArg5, TArg6, TArg7, TArg8, TArg9, TArg10, TArg11, TArg12>(Action<TArg1, TArg2
public void BeginInvoke<TArg1, TArg2, TArg3, TArg4, TArg5, TArg6, TArg7, TArg8, TArg9, TArg10, TArg11, TArg12, TArg13>(Action<TArg
public void BeginInvoke<TArg1, TArg2, TArg3, TArg4, TArg5, TArg6, TArg7, TArg8, TArg9, TArg10, TArg11, TArg12, TArg13, TArg14>(Act
public void BeginInvoke<TArg1, TArg2, TArg3, TArg4, TArg5, TArg6, TArg7, TArg8, TArg9, TArg10, TArg11, TArg12, TArg13, TArg14, TAr
public void BeginInvoke<TArg1, TArg2, TArg3, TArg4, TArg5, TArg6, TArg7, TArg8, TArg9, TArg10, TArg11, TArg12, TArg13, TArg14, TAr

public void Invoke<TArg1>(Action<TArg1> action, TArg1 arg1);
public void Invoke<TArg1, TArg2>(Action<TArg1, TArg2> action, TArg1 arg1, TArg2 arg2);
public void Invoke<TArg1, TArg2, TArg3>(Action<TArg1, TArg2, TArg3> action, TArg1 arg1, TArg2 arg2, TArg3 arg3);
public void Invoke<TArg1, TArg2, TArg3, TArg4>(Action<TArg1, TArg2, TArg3, TArg4> action, TArg1 arg1, TArg2 arg2, TArg3 arg3, TArg
public void Invoke<TArg1, TArg2, TArg3, TArg4, TArg5>(Action<TArg1, TArg2, TArg3, TArg4, TArg5> action, TArg1 arg1, TArg2 arg2, TA
public void Invoke<TArg1, TArg2, TArg3, TArg4, TArg5, TArg6>(Action<TArg1, TArg2, TArg3, TArg4, TArg5, TArg6> action, TArg1 arg1,
public void Invoke<TArg1, TArg2, TArg3, TArg4, TArg5, TArg6, TArg7>(Action<TArg1, TArg2, TArg3, TArg4, TArg5, TArg6, TArg7> action
public void Invoke<TArg1, TArg2, TArg3, TArg4, TArg5, TArg6, TArg7, TArg8>(Action<TArg1, TArg2, TArg3, TArg4, TArg5, TArg6, TArg7,
public void Invoke<TArg1, TArg2, TArg3, TArg4, TArg5, TArg6, TArg7, TArg8, TArg9>(Action<TArg1, TArg2, TArg3, TArg4, TArg5, TArg6,
public void Invoke<TArg1, TArg2, TArg3, TArg4, TArg5, TArg6, TArg7, TArg8, TArg9, TArg10>(Action<TArg1, TArg2, TArg3, TArg4, TArg5
public void Invoke<TArg1, TArg2, TArg3, TArg4, TArg5, TArg6, TArg7, TArg8, TArg9, TArg10, TArg11>(Action<TArg1, TArg2, TArg3, TArg
public void Invoke<TArg1, TArg2, TArg3, TArg4, TArg5, TArg6, TArg7, TArg8, TArg9, TArg10, TArg11, TArg12>(Action<TArg1, TArg2, TAr
public void Invoke<TArg1, TArg2, TArg3, TArg4, TArg5, TArg6, TArg7, TArg8, TArg9, TArg10, TArg11, TArg12, TArg13>(Action<TArg1, TA
public void Invoke<TArg1, TArg2, TArg3, TArg4, TArg5, TArg6, TArg7, TArg8, TArg9, TArg10, TArg11, TArg12, TArg13, TArg14>(Action<TA
public void Invoke<TArg1, TArg2, TArg3, TArg4, TArg5, TArg6, TArg7, TArg8, TArg9, TArg10, TArg11, TArg12, TArg13, TArg14, TArg15>(
public void Invoke<TArg1, TArg2, TArg3, TArg4, TArg5, TArg6, TArg7, TArg8, TArg9, TArg10, TArg11, TArg12, TArg13, TArg14, TArg15,
```

Note: TT recommends using **`BeginInvoke`** instead of **`Invoke`**, as the **`Invoke`** method blocks the calling thread.

When using the **`Dispatcher.AttachUIDispatcher`** method, you can post messages to the event queue using either the **`Invoke`** or **`BeginInvoke`** methods defined in the `TradingTechnologies.TTAPI.Dispatcher` class. You can also perform this task using the methods described in the Thread Binding section.

The following code snippet illustrates how to post a message to the event queue using the `TradingTechnologies.TTAPI.Dispatcher` class.

```
class Test
{
   using TradingTechnologies.TTAPI;

   private WorkerDispatcher m_disp = null;

   public Test()
   {
      m_disp = Dispatcher.AttachWorkerDispatcher();
      m_disp.Run();
   }

   public void PerformSomeWork(string s, int i)
   {
      // ...
   }

   public void PostMessage(string s, int i)
   {
      m_disp.BeginInvoke(PerformSomeWork, s, i);
   }

   // ...
}
```

Threading and Subscriptions

Subscriptions and Event Notifications

After a Dispatcher begins running on a thread, you can begin to create subscriptions on the thread. TT API provides the following subscription classes.

Class	Subscribes for...
PriceSubscription	Market data
TimeAndSalesSubscription	Time & Sales data
TradeSubscription	Orders and fills
SpreadDetailsSubscription	Autospreader spread definitions
CustomerDefaultsSubscription	TT Customer Defaults
FillsSubscription	Historical fills
InstrumentCatalogSubscription	All contracts for a particular product
InstrumentLookupSubscription	A single contract
ProductCatalogSubscription	All products for a specific market
ProductLookupSubscription	A single product corresponding to a specific product key

Creating Subscriptions on the Current Thread

When instantiating these classes, you must provide a reference to a `Dispatcher` in the constructor to indicate which thread TT API uses to fire event notifications for the subscription. For example, TT API fires event notifications for the following price subscription on the current thread.

```
void subscribeForMarketData(Instrument inst)
{
   PriceSubscription priceSub = new PriceSubscription(inst, Dispatcher.Current);
   priceSub.Settings = new
      PriceSubscriptionSettings(PriceSubscriptionType.InsideMarket);
   priceSub.FieldsUpdated += new
      FieldsUpdatedEventHandler(priceSub_FieldsUpdated);
   priceSub.Start();
}

public void priceSub_FieldsUpdated(object sender, FieldsUpdatedEventArgs e)
{
   // process price update
}
```

Creating Subscriptions on a Different Thread

If you want event notifications to fire on a separate thread, simply create a separate thread, attach a dispatcher, and start the subscription, as follows:

```
using TradingTechnologies.TTAPI;

class TTAPIFunctions
{
   private UniversalLoginTTAPI m_apiInstance = null;

   // ...

   public void launchThreads()
   {
      // Launch two Time & Sales subscriptions - each in their own thread

      TS_Sub ps1 = new TS_Sub(m_apiInstance, MarketKey.Cme,
         ProductType.Future, "ES", "Sep13");
      Thread t1 = new Thread(ps1.Start);
      t1.Name = "Thread 1";
      t1.Start();
```

```
      TS_Sub ps2 = new TS_Sub(m_apiInstance, MarketKey.Cme,
        ProductType.Future, "ES", "Jun13");
      Thread t2 = new Thread(ps2.Start);
      t2.Name = "Thread 2";
      t2.Start();
    }

    // ...
}

/// <summary>
/// Instance of the class will be run on a separate thread
/// </summary>
class TS_Sub : IDisposable
{
    /// <summary>
    /// Declare the TTAPI objects
    /// </summary>
    private UniversalLoginTTAPI m_apiInstance = null;
    private WorkerDispatcher m_disp = null;
    private object m_lock = new object();
    private InstrumentLookupSubscription m_req = null;
    private TimeAndSalesSubscription m_tsSub = null;
    private bool m_disposed = false;

    /// <summary>
    /// Declare contract information objects
    /// </summary>
    private MarketKey m_marketKey;
    private ProductType m_productType;
    private string m_product;
    private string m_contract;

    /// <summary>
    /// Primary constructor
    /// </summary>
    public TS_Sub(UniversalLoginTTAPI api, MarketKey mk, ProductType pt,
      string prod, string cont)
    {
      m_apiInstance = api;
      m_marketKey = mk;
      m_productType = pt;
      m_product = prod;
      m_contract = cont;
    }

    /// <summary>
    /// Clean up TTAPI objects
    /// </summary>
    public void Dispose()
    {
      lock (m_lock)
      {
        if (!m_disposed)
        {
          // Detach callbacks and dispose of all subscriptions
          if (m_req != null)
          {
            m_req.Update -= req_Update;
            m_req.Dispose();
            m_req = null;
          }
          if (m_tsSub != null)
          {
            m_tsSub.Update -= tsSub_Update;
            m_tsSub.Dispose();
```

```
            m_tsSub = null;
          }

          // Shutdown the Dispatcher
          if (m_disp != null)
          {
            m_disp.BeginInvokeShutdown();
            m_disp = null;
          }

          m_disposed = true;
        }
      }
    }

    /// <summary>
    /// Create and start the Dispatcher
    /// </summary>
    public void Start()
    {
      // Attach a WorkerDispatcher to the current thread
      m_disp = Dispatcher.AttachWorkerDispatcher();
      m_disp.BeginInvoke(new Action(Init));
      m_disp.Run();
    }

    /// <summary>
    /// Begin work on this thread
    /// </summary>
    public void Init()
    {
      // Perform an instrument lookup
      m_req = new InstrumentLookupSubscription(m_apiInstance.Session,
        Dispatcher.Current,
        new ProductKey(m_marketKey, m_productType, m_product), m_contract);
      m_req.Update += new
        EventHandler<InstrumentLookupSubscriptionEventArgs>(req_Update);
      m_req.Start();
    }

    /// <summary>
    /// Event notification for instrument lookup
    /// </summary>
    public void req_Update(object sender, InstrumentLookupSubscriptionEventArgs e)
    {
      if (e.Instrument != null && e.Error == null)
      {
        // Start a Time & Sales subscription
        m_tsSub = new TimeAndSalesSubscription(e.Instrument, Dispatcher.Current);
        m_tsSub.Update += new EventHandler<TimeAndSalesEventArgs>(tsSub_Update);
        m_tsSub.Start();
      }
      else if (e.IsFinal)
      {
        // Instrument was not found and TT API has given up looking for it
        Console.WriteLine("Cannot find instrument: " + e.Error.Message);
        Dispose();
      }
    }

    /// <summary>
    /// Event notification for Time & Sales updates
    /// </summary>
    public void tsSub_Update(object sender, TimeAndSalesEventArgs e)
    {
      // process the update
      if (e.Error == null)
```

```
        {
          foreach (TimeAndSalesData tsData in e.Data)
          {
            Console.WriteLine("{0} : {1} --> LTP/LTQ : {2}/{3}",
              Thread.CurrentThread.Name, e.Instrument.Name, tsData.TradePrice,
              tsData.TradeQuantity);
          }
        }
      }
    }
```

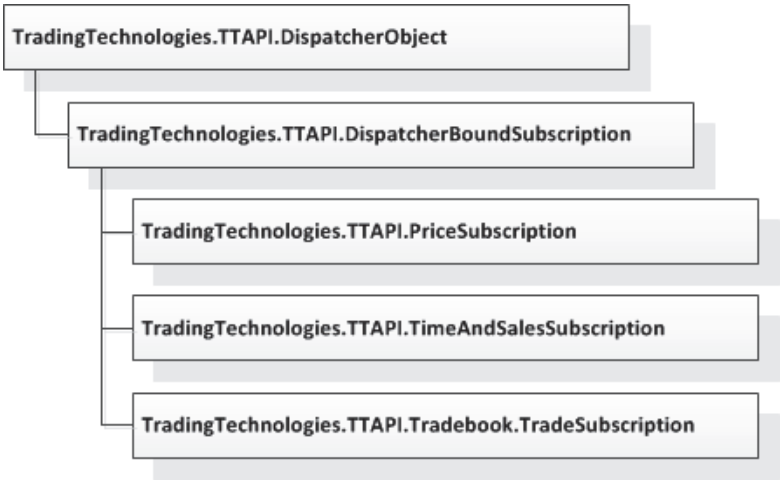
Working with Different Subscription Types

Overview

In terms of threading, there are two types of subscriptions: those that are thread-bound and those that are not. You can modify instances of thread-bound subscriptions only from the thread in which they are created, while you can modify instances of subscriptions that are not thread-bound from any thread in your application.

Thread-Bound Subscriptions

All thread bound subscriptions are derived from the `TradingTechnologies.TTAPI.DispatcherObject` class.



Because you can modify instances of thread-bound classes only from the thread in which you create them, you do not need to use locks to protect against race conditions. Consequently, better performance is achieved.

The TT API offers the following methods to determine whether the ID of the currently executing thread is the same as the ID of the thread on which the subscription was created.

- `TradingTechnologies.TTAPI.DispatcherObject` methods:
  - **`CheckAccess`**, which returns a Boolean value indicating whether the ID of the currently executing thread is the same as the ID of the thread on which the object was created. If this method returns true, you can safely modify the object directly; if it returns false, you must use either the **`Invoke`** or **`BeginInvoke`** method to post a message to the Dispatcher associated with the thread.
  - **`VerifyAccess`**, which throws an exception if the ID of the currently executing thread is not the same as the ID of the thread on which the object was created. If an exception is not thrown, you can safely modify the object directly; if an exception is thrown, you must use either the **`Invoke`** or **`BeginInvoke`** method to post a message to the Dispatcher.
- `TradingTechnologies.TTAPI.Dispatcher` methods:
  - **`CheckAccess`**, which returns a Boolean value indicating whether the ID of the currently executing thread is the same as the ID of the thread on which the object was created. If this method returns true, you can safely modify the object directly; if it returns false, you must use either the **`Invoke`** or **`BeginInvoke`** method to post a message to the Dispatcher associated with the thread.

- **`VerifyAccess`**, which throws an exception if the ID of the currently executing thread is not the same as the ID of the thread on which the object was created. If an exception is not thrown, you can safely modify the object directly; if an exception is thrown, you must use either the **`Invoke`** or **`BeginInvoke`** method to post a message to the Dispatcher.
- **`InvokeRequired`**, which returns a Boolean value indicating whether the ID of the currently executing thread is the same as the ID of the thread on which the object was created. If this method returns false, you can safely modify the object directly; if it returns true, you must use either the **`Invoke`** or **`BeginInvoke`** method to post a message to the Dispatcher associated with the thread.

The following code snippet demonstrates this procedure using the **`DispatcherObject.CheckAccess`** method.

```
private Thread demoThread = null;
private PriceSubscription m_ps;

private void LaunchThread()
{
  this.demoThread = new Thread(new ThreadStart(this.ThreadProcSafe));
  this.demoThread.Start();
}

private void ThreadProcSafe()
{
  this.AccessPriceSubscription();
}

private void AccessPriceSubscription()
{
  // Check if we can modify this object from the current thread
  if (m_ps.CheckAccess())
  {
    //  If so, we can just update the object (m_ps)
  }
  else
  {
    // If not, post a message to the associated Dispatcher
    this.Dispatcher.BeginInvoke(new Action(ThreadProcSafe));
  }
}
```

Note: Following Microsoft's convention, the **`InvokeRequired`** method returns **false** if the ID of the currently executing thread is the same as the ID of the thread on which the control was created, while the **`CheckAccess`** method returns true.

Creating Your Own Thread-Bound Classes

You can also create your own thread bound classes by deriving from the `TradingTechnologies.TTAPI.DispatcherObject` class as follows.

```
using TradingTechnologies.TTAPI;

class MyData : DispatcherObject
{
  MyData(Dispatcher dispatcher)
    : base(dispatcher)
  {
    // ...
  }

  MyData(MyData md)
    : base(md)
  {
    // ...
  }

  // ...
}
```

Choosing a Thread for Thread-Bound Subscriptions

When designing multi-threaded TT API applications, TT recommends that you execute all related tasks for each distinct activity on the same thread to minimize context switching.

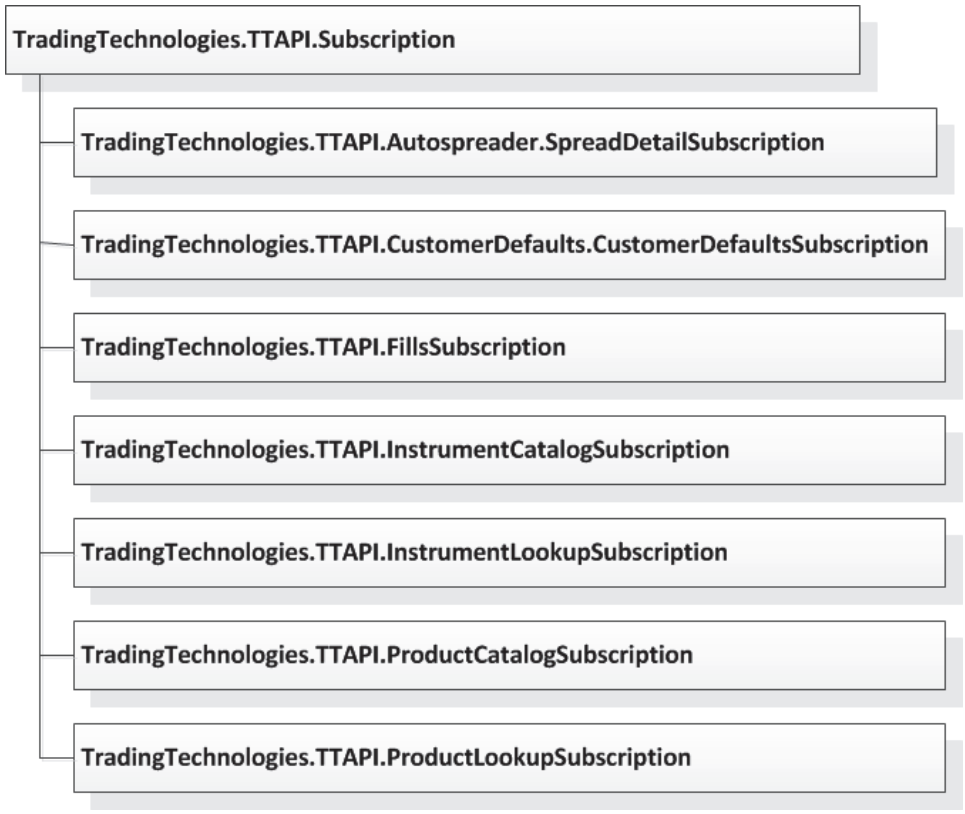
For example, if you are designing a strategy whose inputs comprise the inside market for three Eurex FGBL contracts, you should put the three `PriceSubscription` instances, the `TradeSubscription` instance, and the strategy logic on the same thread. If you put any of the `PriceSubscription` instances and the strategy logic on separate threads, the system would need perform a context switch for each market data update, which could cause thrashing if the frequency of updates is high enough. (Thrashing occurs when a system is overwhelmed by context switching.) If you add another strategy to the same process whose inputs comprise the inside market for two CME ES contracts, you should put these two new `PriceSubscription` instances, the `TradeSubscription` instance, and the strategy logic on a single separate thread.

As a second example, suppose you are designing a WPF application that allows a user to view market data graphically and route orders. Because WPF Controls are thread bound, you should put all of the `PriceSubscription` and `TradeSubscription` instances on the same thread as the WPF Control instances to avoid a context switch on every market data update. Alternatively, you could put all `PriceSubscription` instances on a separate thread and only perform a context switch to update the WPF Controls on a fixed time interval.

Note: Take care not to design applications that perform a context switch on events that occur frequently, such as price updates.

Non-Thread-Bound Subscriptions

All subscriptions that are not thread bound are derived from the `TradingTechnologies.TTAPI.Subscription` class.



To guard against race conditions, the `CustomerDefaultSubscription`, `InstrumentCatalogSubscription` and `ProductCatalogSubscription` classes use a lock to guard its data against race conditions. The `FillsSubscription`, `SpreadDetailSubscription`, `InstrumentLookupSubscription`, and `ProductLookupSubscription` classes do not have any data to protect, so they do not need locks. As such, instances of these classes can be accessed from any thread regardless of the thread on which they were created.

Performance Considerations

Price Subscriptions and Coalescing

With one exception, TT API does not coalesce events that result from subscriptions. TT API can, however, coalesce market data subscription events for instances of the `PriceSubscription` class. Specifically, all market data updates received by the TT API from the corresponding TT Gateway will be coalesced if your application has outstanding notifications for which it has not yet completed processing.

Consider the following market data subscription:

```
void subscribeForMarketData(Instrument inst)
{
   PriceSubscription priceSub = new PriceSubscription(inst, Dispatcher.Current);
   priceSub.Settings = new
      PriceSubscriptionSettings(PriceSubscriptionType.InsideMarket);
   priceSub.FieldsUpdated += new
      FieldsUpdatedEventHandler(priceSub_FieldsUpdated);
   priceSub.Start();
}

public void priceSub_FieldsUpdated(object sender, FieldsUpdatedEventArgs e)
{
   // process price update
}
```

When the TT API receives an inside market update for this contract from the TT Gateway, it posts a message to the event queue to call the **`priceSub_FieldsUpdated`** method. This message includes the updated fields. If the TT API receives more inside market updates for this contract before the call to **`priceSub_FieldsUpdated`** has completed, it coalesces the data. For example, assume the first inside market data update contains:

FieldId	Value
BestBidPrice	103.18
BestBidQuantity	10
BestAskPrice	103.20
BestAskQuantity	15

Now assume that the TT API receives the following inside market updates from the TT Gateway before the call to the **`priceSub_FieldsUpdated`** method has completed:

Update #	FieldId	Value
1	BestBidQuantity	15
2	BestBidQuantity	2
	BestAskQuantity	44
3	BestBidPrice	103.19
	BestBidQuantity	35

The data contained in the next message posted to the event queue to call the **`priceSub_FieldsUpdated`** method contains:

FieldId	Value
BestBidPrice	103.19
BestBidQuantity	35
BestAskQuantity	44

Therefore, the longer it takes your application to process an update, the more coalescing is likely to occur.

Note: TT API coalesces all fields, including **`LastTradedPrice`** and **`LastTradedQuantity`**. If you need an uncoalesced trade data feed, you should use the `TimeAndSalesSubscription` class

Order Routing

TT API provides two methods for routing orders: from the `Session` object or from a `TradeSubscription` object. The following example shows how to route an order using the **`TradeSubscription.SendOrder`** method.

```
public bool routeOrder(TradeSubscription ts, OrderProfile op)
{
  if (!ts.SendOrder(op))
  {
     Console.WriteLine("Send Order failed : {0}", op.RoutingStatus.Message);
     return false;
  }
  else
  {
     Console.WriteLine("Send Order succeeded.");
     return true;
  }
}
```

Because `TradeSubscription` instances are thread bound, you must call the **`SendOrder`** method from the thread on which the instance was created. Calling this method from a different thread throws an exception.

Note: For optimal performance, you should set the **`OwnOrdersOnly`** property to true in the **`TradeSubscription`** constructor if possible. Doing so means that a `TradeSubscription` instance tracks and reports updates only for orders that were submitted through it.

You can also route orders by calling the **`UniversalLoginTTAPI.Session.SendOrder`** or **`XTraderModeTTAPI.Session.SendOrder`** methods as follows.

```
public bool routeOrder(UniversalLoginTTAPI ttapi, OrderProfile op)
{
  if (!ttapi.Session.SendOrder(op))
  {
     Console.WriteLine("Send Order failed : {0}", op.RoutingStatus.Message);
     return false;
  }
  else
  {
     Console.WriteLine("Send Order succeeded.");
     return true;
  }
}
```

Because `UniversalLoginTTAPI` and `XTraderModeTTAPI` instances are not thread bound, you can call this method from any thread. For optimal performance, TT recommends using this method when routing orders from multiple threads.

Send Us Your Comments

Writing Multithreaded TT API Applications

Version 7.2.X

Trading Technologies® welcomes your comments and suggestions on the accuracy and usefulness of this publication. Your input is important and valuable in revising our documentation and helps ensure a constantly improving level of quality.

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Are the examples correct? Do you need more examples?
- Which features did you find particularly useful?
- What did you like most about this manual or document?

If you encounter any errors in this document or would like to share other suggestions you might have for improving this document, send comments to: documentation.dept@tradingtechnologies.com.

If possible, please indicate the chapter, section, and page number relevant to your feedback.