X_TRADER API Tips and Tricks

- A course by Trading Technologies -

Intended Audience

- Prior Development Experience with C++, C#, or Visual Basic
- Familiar with XTAPI
- Familiar with Trading Application logic
- This is not an Introduction!

Agenda

XTAPI Overview

- **XTAPI is a collection of objects and interfaces that provide access to the TT environment through custom applications.**

- **XTAPI objects provide real-time access to**
  - Market and Implied Prices, including Depth
  - Send, Modify, and Cancel Orders
  - Risk Administration and Back Office tasks
  - Market Browsing and Contract Specifications
  - Receive and process Fill records

XTAPI Overview

- **XTAPI uses Microsoft's COM technology**
- **Any COM-enabled language can be used to develop XTAPI applications**
  - Examples: C++, Visual Basic
- **.Net languages are compatible but not native**
  - Examples: C#, Visual Basic.Net

Agenda

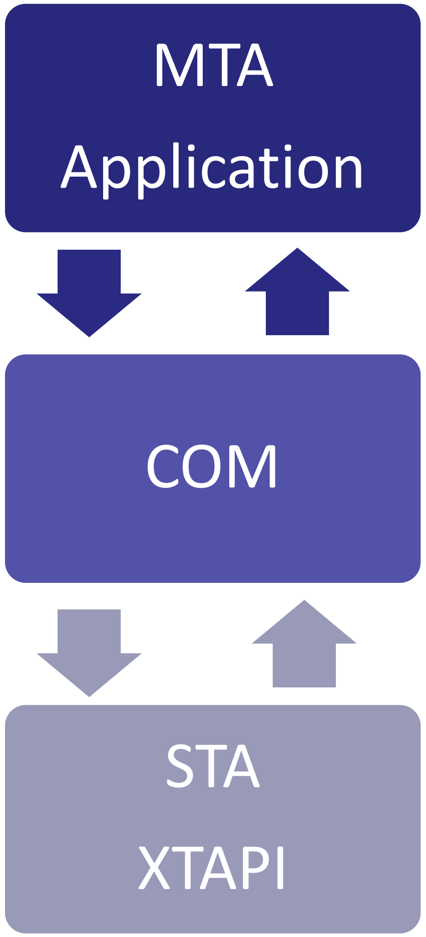
XTAPI is STA

XTAPI is a Single-Threaded Apartment (STA) model.

If you are able to develop your application with the STA model, you should. Because XTAPI is STA, you will not pay any marshalling or thread context switching penalties.

TT does not recommend the Multi Threaded Apartment (MTA) Model because your application will incur thread context switching with each method call to XTAPI.

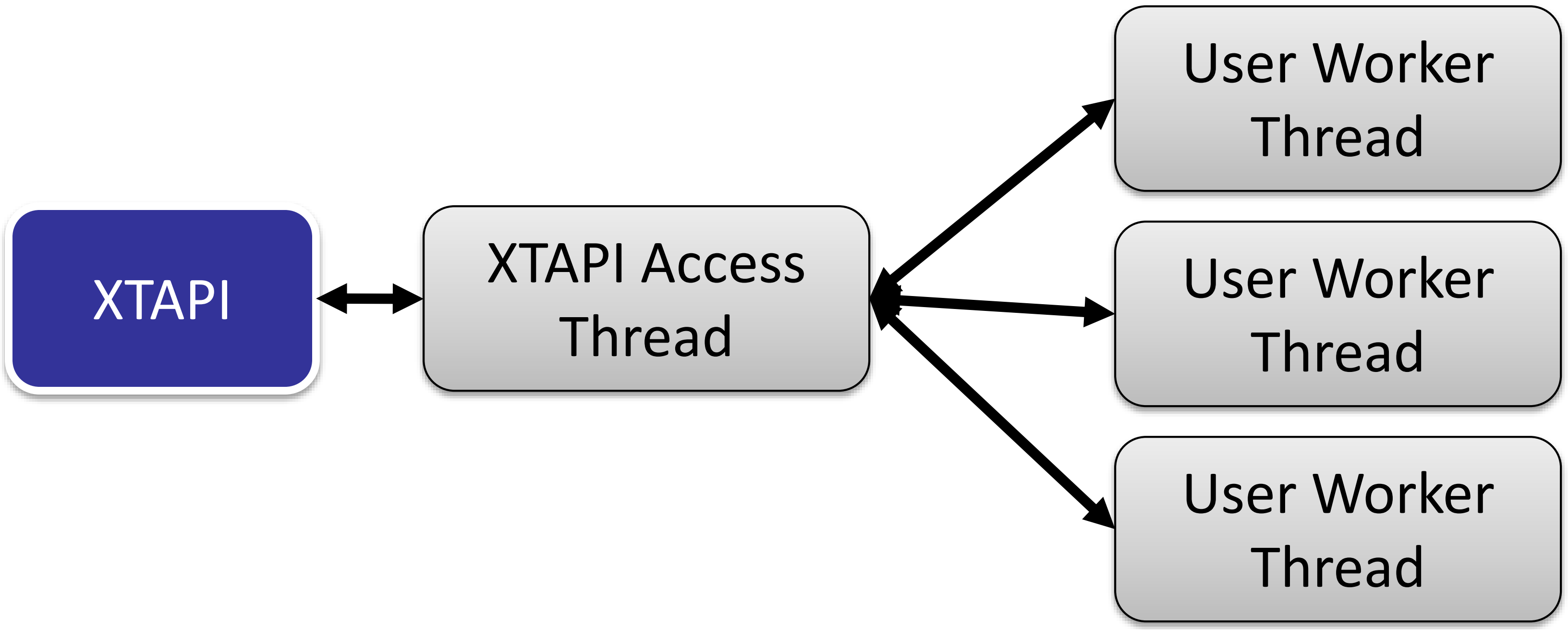
Avoiding the COM Barrier



A thread context switch will occur whenever a method or property of XTAPI is called from your MTA application.

Multi-Threaded Applications

To develop a multi-threaded XTAPI application, you should still select the STA model. Create a single thread that will act as the entry point to XTAPI, and have all other threads in your application access XTAPI through this single thread



Get Properties

The Get() method causes to XTAPI allocate a VARIANT and return it to your application for extraction.

```
private void m_TTInstrNotify_OnNotifyUpdate(XTAPI.TTInstrNotify pNotify,
   XTAPI.TTInstrObj pInstr)
{
    if (pInstr == null)
        return;
    string instrLTP = (string)pInstr.get_Get("Last");  1
    double todaysHigh = (double)pInstr.get_Get("High#");  2
    int netPos = (int)pInstr.get_Get("NetPos");  3
    string nativePL = (string)pInstr.get_Get("PL^");  4
}
```

4 VARIANTS have been created!

Avoiding the COM Barrier

To optimize your code, condense the number of VARIANT objects that need to be created

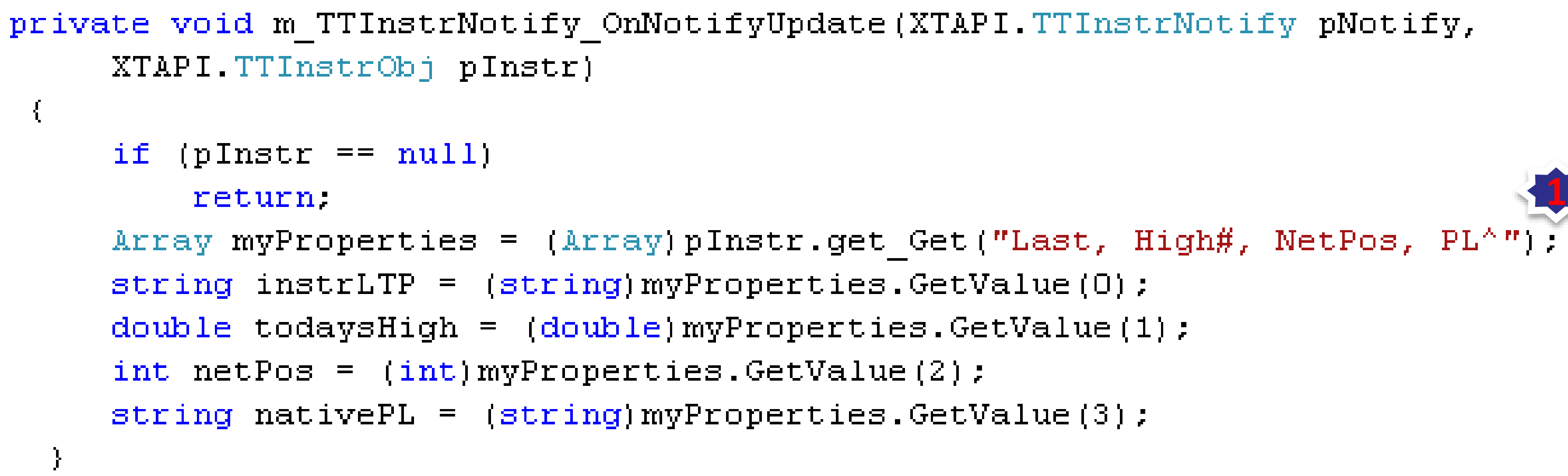
Optimally, we would need only a single VARIANT object to be created, with each of our data members contained within it

Compound Get Properties

This optimization is accomplished by using Compound Get Properties

Compound Get Properties request multiple XTAPI properties within a single method call. This eliminates the need for XTAPI to repeatedly allocate individual VARIANT objects

Example: Compound Get Properties



```
private void m_TTInstrNotify_OnNotifyUpdate(XTAPI.TTInstrNotify pNotify,
     XTAPI.TTInstrObj pInstr)
 {
     if (pInstr == null)
         return;
     Array myProperties = (Array)pInstr.get_Get("Last, High#, NetPos, PL^");
     string instrLTP = (string)myProperties.GetValue(0);
     double todaysHigh = (double)myProperties.GetValue(1);
     int netPos = (int)myProperties.GetValue(2);
     string nativePL = (string)myProperties.GetValue(3);
  }
```

A single VARIANT is created to hold all of the property values. Data extraction into typed values occurs in native code.

Agenda

1. XTAPI Overview
2. The COM Barrier
3. Data Conversions
4. Market Data Updates
5. Filters
6. Rapid Fill Delivery
7. Properties
8. Implied Prices
9. Event Notifications
10. Order Sets
11. Series Keys

Ticks != Points != X_TRADER Display

XTAPI provides specific data types for many properties that allow you to request values in different formats.

Price Formats
X_TRADER String Format
Ticks
Currency
Delta

Data Types
Integer
Decimal
String

Let XTAPI handle the data conversions.

Bad Example

The following code snippet shows the extraction of the Best Bid value in X_TRADER string format, and then converts that price to a Tick Value.

```
void m_notify_OnNotifyUpdate(TTInstrNotify pNotify, TTInstrObj pInstr)
{
    if (pInstr == null)
        return;
    // extract the Best Bid in X_TRADER String format
    string myXTPrice = (string)pInstr.get_Get("Bid");
    int myXTPriceAsInt = Convert.ToInt32(myXTPrice);
    // extract the Tick Size for this contract
    int tickSize = (int)pInstr.get_Get("TickIncrement");

    // Overly simplified conversion from String to Tick
    int tickPrice = myXTPriceAsInt / tickSize;


    ...
}
```

This requires the user to add their own data conversion code

Use XTAPI to do your data conversion

```
void m_notify_OnNotifyUpdate(TTInstrNotify pNotify, TTInstrObj pInstr)
{
    if (pInstr == null)
         return;
    // extract the Best Bid in TICKS
    int myTickPrice = (int)pInstr.get_Get("Bid&");

}
```

Specifies Ticks Format

Using Ticks is recommended

- XTAPI natively works with prices in Ticks
- Modifying your application to use tick prices will eliminate the need to convert data to and from different formats both in your application as well as in XTAPI

Agenda

Inside Market vs Depth Updates

To subscribe for Inside Market updates: OnNotifyUpdate

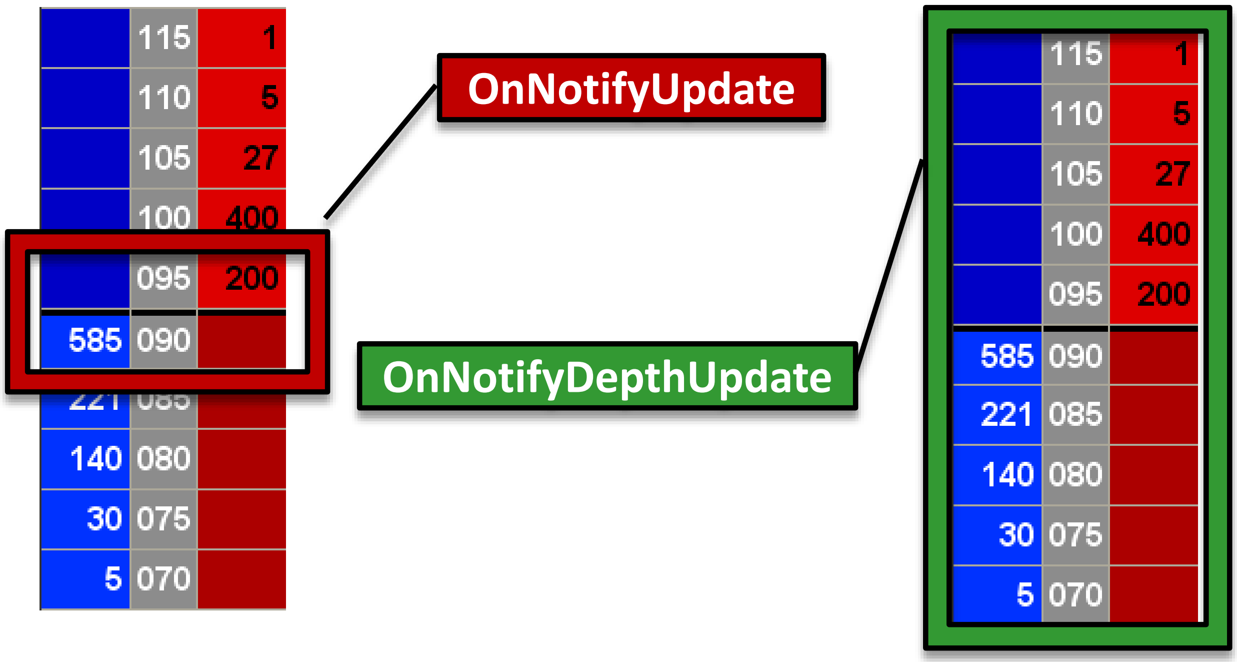
```
// On Notify Update informs us that values have changed for this Instrumment, including Inside Market
m_notify.OnNotifyUpdate +=
    new _ITTInstrNotifyEvents_OnNotifyUpdateEventHandler(m_notify_OnNotifyUpdate);
```

To subscribe for Depth updates: OnNotifyDepthUpdate

```
// On Notify Depth Update informs us that Depth of Book values have changed (Price or Quantity)
m_notify.OnNotifyDepthData +=
    new _ITTInstrNotifyEvents_OnNotifyDepthDataEventHandler(m_notify_OnNotifyDepthData);
```

Depth + Inside Market

A market depth update will include any inside market updates



Depth vs. Inside Market Example

- **Assume you are writing an automated Market Making application.**
  - Per your Market Making agreement with the Exchange, your quoting orders must be within 2 levels of the Inside Market
  - When the Inside Market moves, you **may** want to re-quote your order. The decision is based on liquidity
    - If there is sufficient liquidity, you can quote at the Inside Market
    - If there is not, you want to quote up to 2 levels away, in an attempt to avoid getting filled

Depth vs IM Example…

- Based on the application requirements, you need only to take an action if the Inside Market values change

- Therefore you can subscribe to Inside Market changes and only request Depth when needed

- This will result in fewer event callbacks, increasing application performance

Depth vs IM Example…

Summary of Application Logic

The application subscribes to the **OnNotifyUpdate** event in order to receive a notification when the **Inside Market** values change

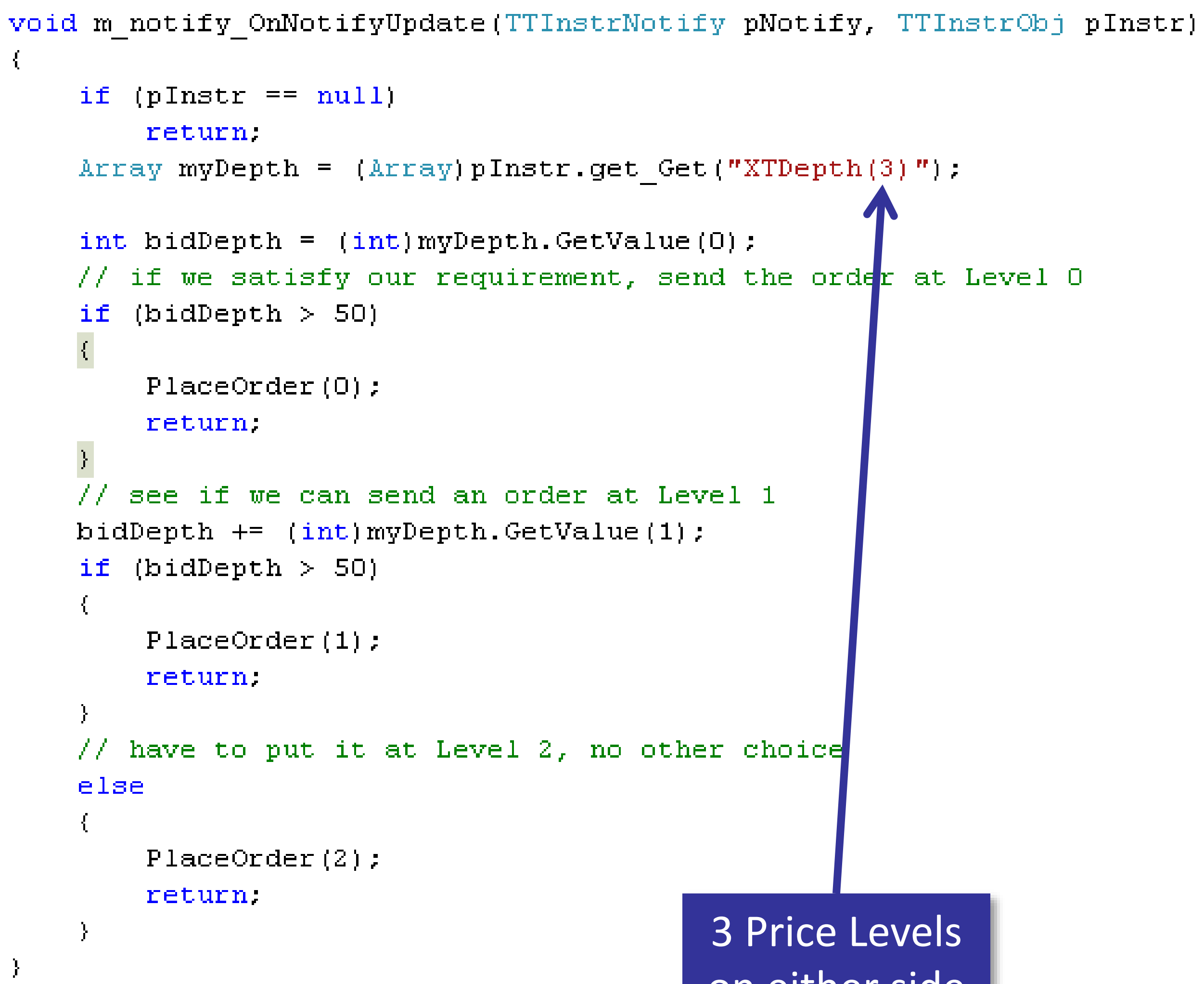
When the Inside Market changes, the **OnNotifyUpdate** event will be fired. Within your event handler method, request Market Depth

Based on the market making strategy, orders will be placed in the market:

If…Else if…	**Then….**
Level 0 quantity >= 50	Place order at Level 0
Level 0 + Level 1 >= 50	Place order at Level 1
Neither of the above are true	Place order at Level 2

Depth vs IM Example…

This will result in significantly fewer callbacks. It also only gives you a depth snapshot when there has been a change to the **Inside Market**, eliminating superfluous notifications of outside quoting, and improving the overall efficiency of the application



```
void m_notify_OnNotifyUpdate(TTInstrNotify pNotify, TTInstrObj pInstr)
{
    if (pInstr == null)
        return;
    Array myDepth = (Array)pInstr.get_Get("XTDepth(3)");

    int bidDepth = (int)myDepth.GetValue(0);
    // if we satisfy our requirement, send the order at Level 0
    if (bidDepth > 50)
    {
        PlaceOrder(0);
        return;
    }
    // see if we can send an order at Level 1
    bidDepth += (int)myDepth.GetValue(1);
    if (bidDepth > 50)
    {
        PlaceOrder(1);
        return;
    }
    // have to put it at Level 2, no other choice
    else
    {
        PlaceOrder(2);
        return;
    }
}
```

3 Price Levels on either side of the Market

Agenda

Filter Updates for the values you want

When using an Instrument Notify (TTInstrNotify) object to receive event callbacks, you will receive the event when **any** of the instrument's properties change

As an optimization, you can set a Notify Filter that will only fire the event callbacks when the specified properties change

This is accomplished by using the UpdateFilter property

```
m_TTInstrNotify.UpdateFilter(m_TTInstrObj, "Last, Bid, Ask");
```

Now only changes to the Last Traded Price, the Best Bid, or the Best Ask will result in the event being fired.

Back to the Depth vs IM Example…

- Our Market Making example can be further optimized. Currently we are receiving the OnNotifyUpdate event when **any** of the Inside Market values change (Price, Quantity, High, Settle, etc…)

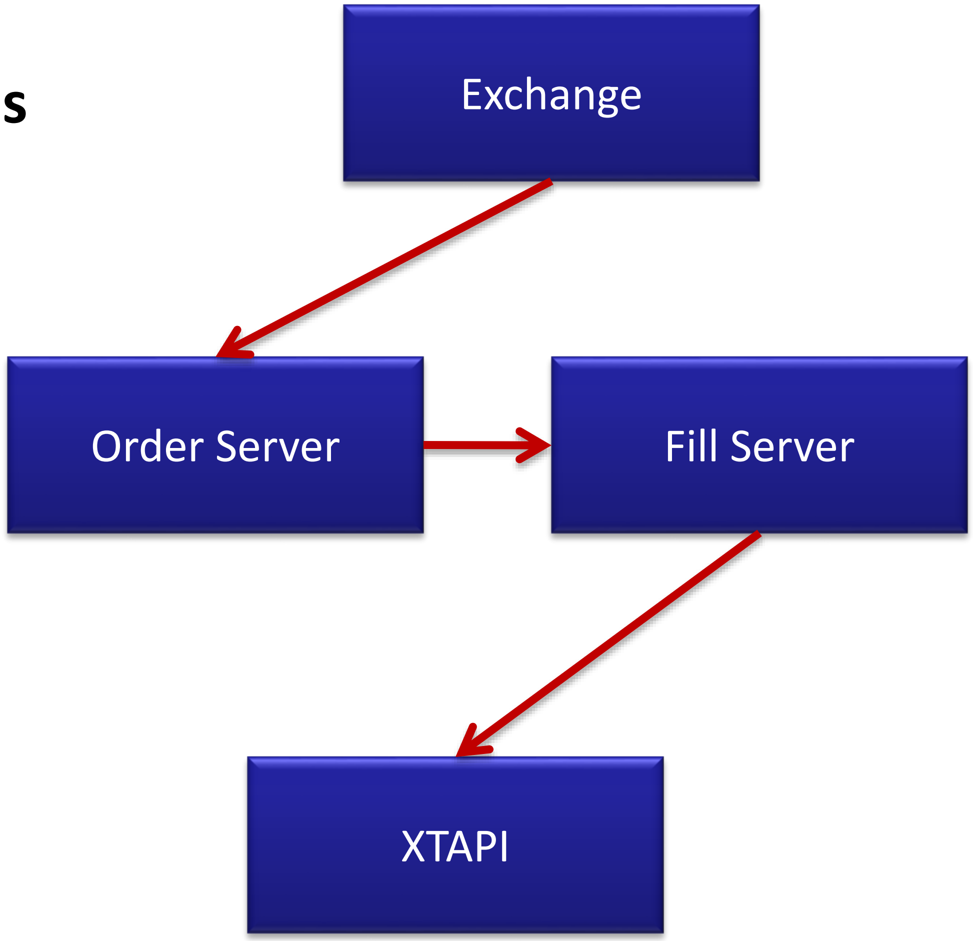
- By setting a Notify Filter, we can limit the event callbacks to fire **only** when the Inside Market Price changes, ignoring any other updates

```
m_notify.UpdateFilter = "Last";
```

Agenda

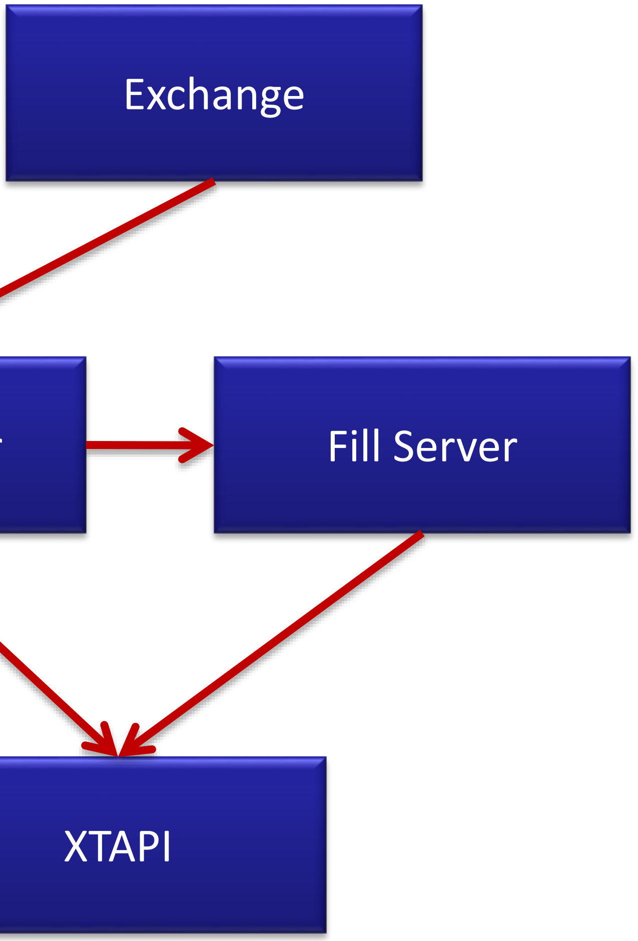
Rapid Fill Delivery

When Rapid Fill Delivery is OFF (Default)



Rapid Fill Delivery

When Rapid Fill Delivery is ON



XTAPI accepts the first fill it receives

Rapid Fill Delivery Caveats

XTAPI's Order Sets listen to Fill notifications from the Order Server, regardless of the Rapid Fill Delivery setting. This can lead to "in flight" issues.

For example, assume you have Rapid Fill Delivery set to **TRUE**, and you have a single Order Set containing two working orders.

Your application receives a Full Fill Record from the Fill Server. In your event handler method, you query for the number of orders contained within the Order Set.

The Order Set (may) report back that it contains **2** orders. This seems like an error because your Fully Filled order should no longer exist.

Rapid Fill Delivery Caveats…

It’s not an error.

The Order Set will eventually correct itself – when the Order Server processes the Fill Record and sends the notification to your application. This “in flight” issue must be considered when deciding whether to utilize Rapid Fill Delivery.

Rapid Fill Delivery Caveats…

These properties will not be available from the Fill Record when **Rapid Fill Delivery** is **ON**:

Ex:OrderNo
OrderNo
FillKey

BrokerTec *must* use Rapid Fill Delivery

On BrokerTec, during the Workup Phase, the exchange will not send a Fill Notification until the Workup Phase is complete. The Fill records are held at the Fill Server. To get around this, turn on **Rapid Fill Delivery**, which will force the Fill Receipts to be sent from the Order Server.

You will not receive fill notifications until after the Workup Phase is over **unless** you **enable** Rapid Fill Delivery.

Rapid Fill Delivery Usage Guidelines

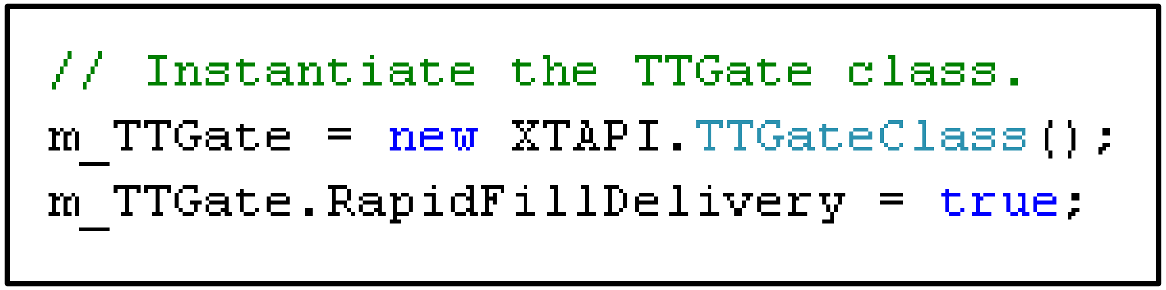
In General…

Turn RFD On if…	**Turn RFD Off if…**
Responding to Fills (e.g. hedging a spread) is the prime function of your application	Accurate Order Book management is the prime function of your application

The determination of whether to **enable** or **disable** Rapid Fill Delivery should be based on the requirements of the application. There is no simple "Yes" or "No" answer.

Rapid Fill Delivery

Now that the costs and benefits of Rapid Fill Delivery are clear, if you decide to enable them in your application, here is how:



```
// Instantiate the TTGate class.
m_TTGate = new XTAPI.TTGateClass();
m_TTGate.RapidFillDelivery = true;
```

Agenda

1. XTAPI Overview
2. The COM Barrier
3. Data Conversions
4. Market Data Updates
5. Filters
6. Rapid Fill Delivery
7. Properties
8. Implied Prices
9. Event Notifications
10. Order Sets
11. Series Keys

Use XTAPI properties

Many users try to track market and order status values themselves.
XTAPI does it for you.

The following slides illustrate several Order Set properties that will provide accurate position information.

Use XTAPI properties – Buy Position

NetPos – Current net position based on fills
BuyPos – Quantity bought based on fills

Example:

You have a single buy order in the market with an order quantity of 12. You receive a partial fill of 8 for the order

OrderSet.NetPos is 8
OrderSet.BuyPos is 8
OrderSet.SellPos is 0

Use XTAPI properties – Sell Position

NetPos – Current net position based on fills
SellPos – Quantity sold based on fills

Example:

You have a single sell order in the market with an order quantity of 18. You receive a partial fill of 11 for the order

OrderSet.NetPos is -11
OrderSet.BuyPos is 0
OrderSet.SellPos is 11

Combined Position Example

Example:

You submit a single buy order with an order quantity of 12.
You receive a partial fill of 8 for the order

You submit a single sell order with an order quantity of 18.
You receive a partial fill of 11 for the order

OrderSet.NetPos is -3
OrderSet.BuyPos is 8
OrderSet.SellPos is 11

OrderSet: BuyCnt / SellCnt

BuyCnt – returns the number of individual buy orders
SellCnt – returns the number of individual sell orders

Example:

There are three buy orders in the market, with order quantities of 5, 12, and 17

There are two sell orders in the market, with order quantities of 2 and 8

OrderSet.BuyCnt = 3
OrderSet.SellCnt = 2

OrderSet: BuyWrk / SellWrk

BuyWrk – returns the aggregate quantity of buy orders
SellWrk – returns the aggregate quantity of sell orders

Example:

There are three buy orders in the market, with order quantities of 5, 12, and 17

There are two sell orders in the market, with order quantities of 2 and 8

OrderSet.BuyWrk = 34
OrderSet.SellWrk = 10

Average Cost of Buys / Sells

This slide assumes that your Order Set contains orders and fills for a single Instrument

Average Cost of Buys (in Ticks) = BuyTicks / BuyPos
Average Cost of Sells (in Ticks) = SellTicks / SellPos

To determine the average cost of your buy orders contained within this Order Set, divide OrderSet.BuyTicks by OrderSet.BuyPos

To determine the average cost of your sell orders contained within this Order Set, divide OrderSet.SellTicks by OrderSet.SellPos

Agenda

1. XTAPI Overview
2. The COM Barrier
3. Data Conversions
4. Market Data Updates
5. Filters
6. Rapid Fill Delivery
7. Properties
8. Implied Prices
9. Event Notifications
10. Order Sets
11. Series Keys

Implied Price Engine

By default, XTAPI has the Implieds Engine turned OFF

By default, RTD has the Implieds Engine ON

To turn XTAPI's Implied Engine ON, the Instrument object's *CalculateTTImplieds* property should be set to TRUE (1)

```
pInstr.CalculateTTImplieds = 1;
```

Exchange vs TT Implieds

XTAPI's Implied Engine will calculate implied prices for markets when the Exchange does not disseminate its own implied prices.

Turning the Implied Engine ON or OFF will not affect Exchange-disseminated implieds prices, only XTAPI-generated implieds.

Certain exchanges (e.g. LIFFE) disseminate their own Implied prices. In this case, turning the XTAPI Implied Engine ON/OFF has no impact.

Implied Depth?

- The XTAPI Implied Engine **does not** calculate Implied Depth.
- If the Exchange disseminates Implied Depth, XTAPI will deliver this data to you.
- Note that X_TRADER does not display Implied Depth. The only way to receive Exchange disseminated Implied Depth, in TT, is through XTAPI.

Direct versus Implied Price Fields

To view Direct Prices only:

	Ask	AskQty
BidQty	Bid	

To view Implied (TT Calculated or Disseminated) Prices only:

	IAsk	IAskQty
IBidQty	IBid	

Merging Implieds

To view the "Best" Price, whether Direct or Implied:

	MIAsk	**MIAskQty**
MIBidQty	**MIBid**	

CalculateTTImplieds has no effect on Merging Implied and Direct quantities. However, only Exchange-disseminated Implied quantities will be merged if CalculateTTImplieds is OFF.

Merging Implieds Examples

- When the **Implied** Bid is the Best Bid:

  MIBid = IBid
  MIBidQty = IBidQty

- When the **Direct** Bid is the Best Bid:

  MIBid = Bid
  MIBidQty = BidQty

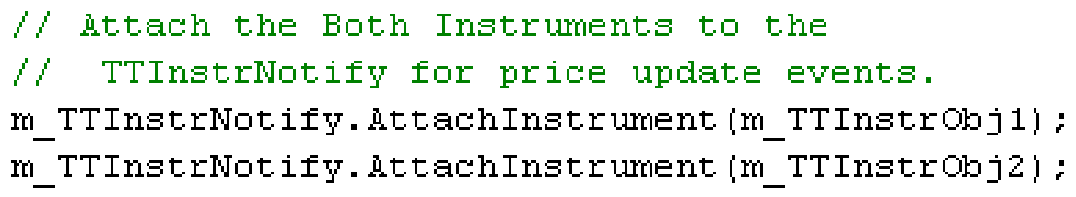
- When **Implied Bid** and **Direct Bid** prices are equal:

  MIBid = Bid
  MIBid = IBid
  MIBidQty = BidQty + IBidQty

Agenda

InstrNotify

You can attach multiple Instruments to a single Instrument Notify object:



```
// Attach the Both Instruments to the
//  TTInstrNotify for price update events.
m_TTInstrNotify.AttachInstrument(m_TTInstrObj1);
m_TTInstrNotify.AttachInstrument(m_TTInstrObj2);
```

Now both instruments will use the same event handler

InstrNotify

Benefits

Could result in less code

Single point of processing

Less events = less context switching

Risks

Processing one contract may lead to missed messages

Lots of IF statements and comparisons

OrderProfile

A TTOrderProfile object is used to populate the attributes of an order. A new order is created by calling the OrderSet.SendOrder() method, passing the Order Profile object as a parameter.

Some things to keep in mind about Order Profiles:

- Is Write-Only

- To query properties of a TTOrderProfile object, use GetLast instead of Get.

- The object's properties are not fully created until the OrderProfile is converted to a TTOrderObj by calling ***TTOrderSet::SendOrder(myOrderProfile);***

TTHotKeyNotify – The Lost Object

You can use the **TTHotKeyNotify** object to receive an event callback when a key combination is pressed. You can map XTAPI functionality to this event

For example, users can create a Hot Key that will fire an event when the Space Bar is pressed. This would be a quick and convenient way to implement a **Delete All Orders** panic button

WARNING! The Hot Key is registered with Windows.
Therefore any use of the key outside the application will result in your application receiving the callback and possibly taking action.

Agenda

1. XTAPI Overview
2. The COM Barrier
3. Data Conversions
4. Market Data Updates
5. Filters
6. Rapid Fill Delivery
7. Properties
8. Implied Prices
9. Event Notifications
10. Order Sets
11. Series Keys

Order Set - Overview

- XTAPI allows the segregation of working orders and fills into logical groupings called Order Sets

- Properties such as P&L, Net Position, Buy Count, etc… are calculated at the Order Set level

- Order Status events fire from Order Sets

- There is always at least one Order Set, but a user can create more

Too many Order Sets

While there is no hard-coded limit to the number of Order Sets a user may create, you should understand that each Order Set comes with a cost.

When a Fill Record or other Update is received by XTAPI, it must evaluate **each** Order Set to determine if an Order Set Update notification is required to be sent to your application.

The more Order Sets that need to be evaluated, the more performance will be impacted.

Be conscious of the cost incurred with every Order Set that you create.

Using multiple Order Sets

May be advantageous because:

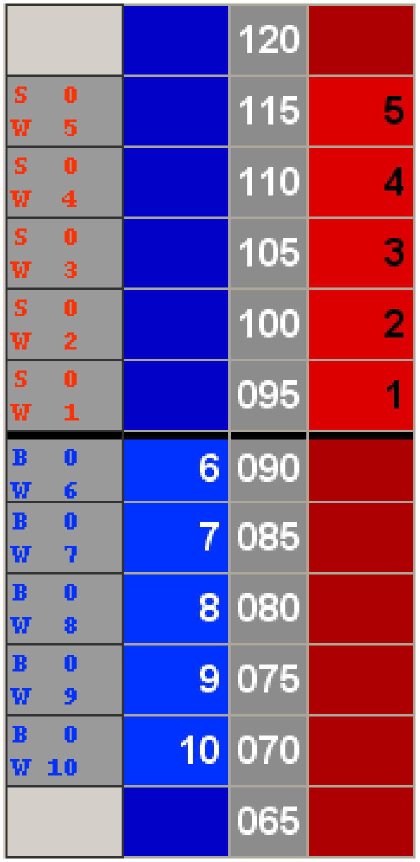
- You can logically group orders, such as buy orders in one order set and sell orders in another.

- For each Order Update, every order in the Order Set must be evaluated. By limiting the number of orders in each Order Set, you may experience performance gains.

Orders in Order Sets are Ordered

When requesting the list of orders contained within an Order Set the returned array is already ordered such that the greater the index, the further away from the Inside Market.

Working orders within an Order Set are ordered such that Working Sells come before Working Buys



		120	
S 0 W 5		115	5
S 0 W 4		110	4
S 0 W 3		105	3
S 0 W 2		100	2
S 0 W 1		095	1
B 0 W 6	6	090	
B 0 W 7	7	085	
B 0 W 8	8	080	
B 0 W 9	9	075	
B 0 W 10	10	070	
		065	

Orders in OrderSet are Ordered

A simple example:

If you have 10 working orders, and the first 5 are Sells, to get the Buy order closest to the inside market:

```
TTInstrObj myInstr = (TTInstrObj)m_TTOrderSet[5];
```

A more dynamic example:

Use XTAPI properties to determine the number of working Sell orders, And then offset the array to jump to the first Buy order:

```
int workingSells = (int)m_TTOrderSet.get_Get(“WrkSell”);
TTInstrObj myFirstBuy = (TTInstrObj)m_TTOrderSet[workingSells];
```

Orders in OrderSet are Ordered

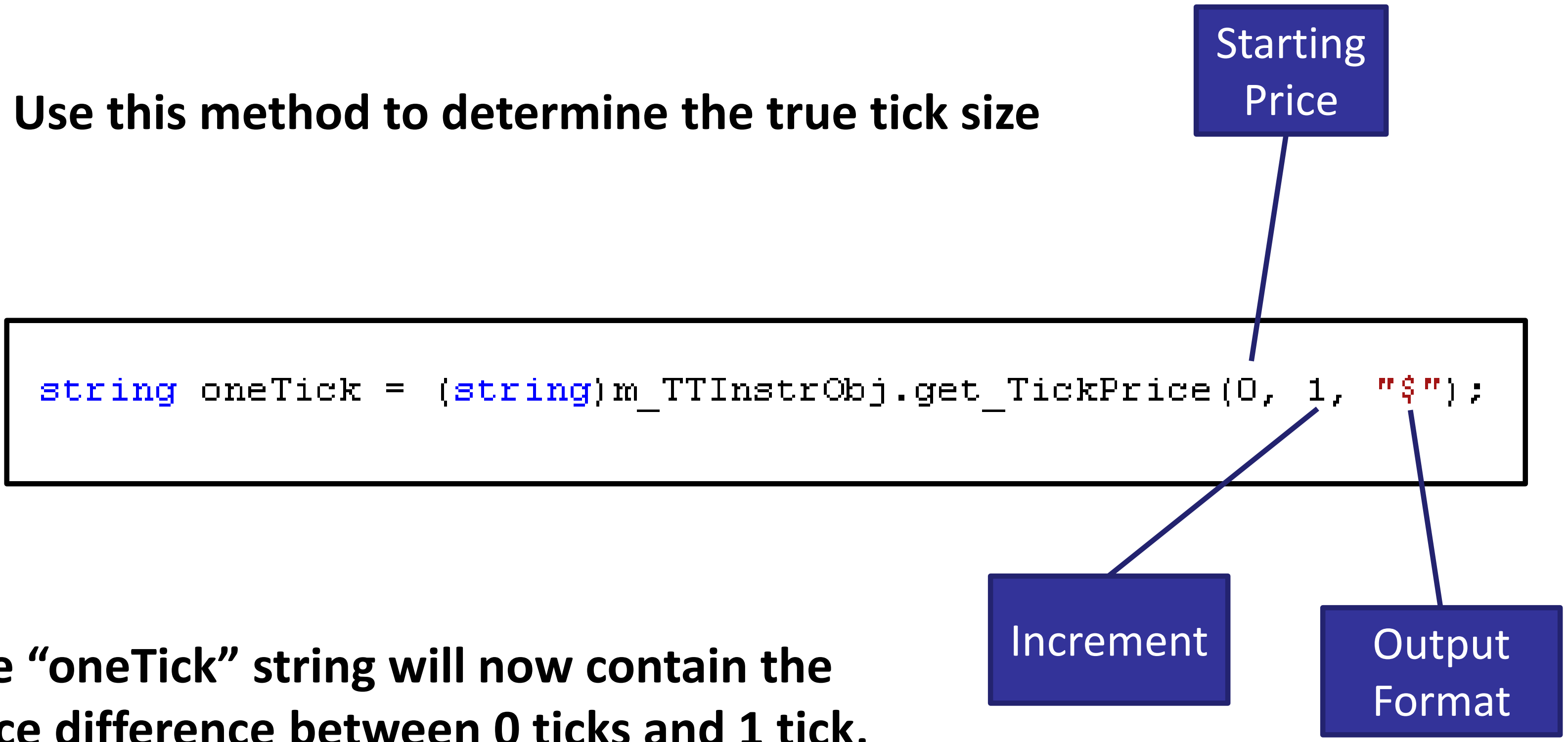
For a single order on either side of the market:

```
TTInstrObj mySell = (TTInstrObj)m_TTOrderSet[0];
TTInstrObj myBuy = (TTInstrObj)m_TTOrderSet[1];
```

With only a single Buy and single Sell in the market, the Sell order will **always** be the first in the array, and the Buy order the second.

TickPrice

Use this method to determine the true tick size



```
string oneTick = (string)m_TTInstrObj.get_TickPrice(0, 1, "$");
```

The “oneTick” string will now contain the price difference between 0 ticks and 1 tick, effectively giving you the contract’s tick size.

Iterating an Order Set

Warning!
The Order Set can change while you are iterating

If you must iterate through the Order Set, use the .Net "**foreach**" instead of the C-style "**for**"

The "**for**" loop, relying on an index to move through a collection, must first load all of the collection members into memory. "**foreach**" implements The IEnumerable interface, which is a native construct in .Net allowing the iteration through a collection without having to load the contents of the collection in memory.

With **foreach**, you can catch a specific exception – **System.InvalidOperationException** – that will occur if the collection is modified during your processing.

Use Ticks to specify price

Internally, XTAPI will convert all prices to Ticks. Specifying a tick price for an order will save the time spent with a conversion.

```
TTOrderProfile profile = new TTOrderProfileClass();
profile.Instrument = m_instr;
profile.Customer = customerCombo.SelectedItem.ToString();
profile.Set("BuySell", buySell);
profile.Set("Qty", orderQtyCombo.Value.ToString());

int myTickPrice = (int)m_instr.get_Get("Last&"); // get LTP in Ticks
int tickSize = (int)m_instr.get_TickPriceEx(0, enumRoundPriceType.ROUND_UP,
    1, "&"); // get the tick size - diff between 0 and 1 tick
myTickPrice += tickSize;  //increment our LTP by 1 tick

profile.Set("Limit&", myTickPrice); // set the Limit price to my new Tick Price

int numSent = m_orderSet.get_SendOrder(profile); // send the order
if (numSent != orderQtyCombo.Value)
{
    MessageBox.Show("You requested " + orderQtyCombo.Value.ToString() +
        " but only " + numSent.ToString() + " orders were placed.");
}
```

Agenda

1. XTAPI Overview
2. The COM Barrier
3. Data Conversions
4. Market Data Updates
5. Filters
6. Rapid Fill Delivery
7. Properties
8. Implied Prices
9. Event Notifications
10. Order Sets
11. Series Keys

Series Keys

A Series Key is a numeric identifier for a Financial Instrument within the TT environment.

Notes:

•Series keys are ***guaranteed*** to be valid only within a single exchange session

•If your application stores Series Keys, you should re-request the key at the beginning of each exchange session.

Working with Series Keys

To request the series key, open an instrument by specifying the Gateway / Product / Product Type / Contract.

Then request the Series Key from the Instrument object:

```
m_TTInstrObj = new TTInstrObj();
m_TTInstrObj.Exchange = "CBOT";
m_TTInstrObj.ProdType = "SPREAD";
m_TTInstrObj.Product = "ZN";
m_TTInstrObj.Contract = "Calendar: 1xZN DEC09:-1xZN Mar10";

m_TTInstrNotify.AttachInstrument(m_TTInstrObj);

m_TTInstrObj.Open(0);

/****/

string mySeriesKey = m_TTInstrObj.get_Get("SeriesKey");
```

Working with Series Keys

You can also open an instrument with a Series Key, and then extract the Gateway / Product / Product Type / Contract properties:

```
m_TTInstrObj = new TTInstrObj();
m_TTInstrObj.SeriesKey = "12345679034321"; // DEMO ONLY!

m_TTInstrNotify.AttachInstrument(m_TTInstrObj);

m_TTInstrObj.Open(0);

Array contractSpecs =
    (Array)m_TTInstrObj.get_Get("Exchange,ProdType,Product,Contract");
string myExchange = (string)contractSpecs.GetValue(0);
string myProdType = (string)contractSpecs.GetValue(1);
string myProduct  = (string)contractSpecs.GetValue(2);
string myContract = (string)contractSpecs.GetValue(3);
```

Example of a previously extracted Series Key