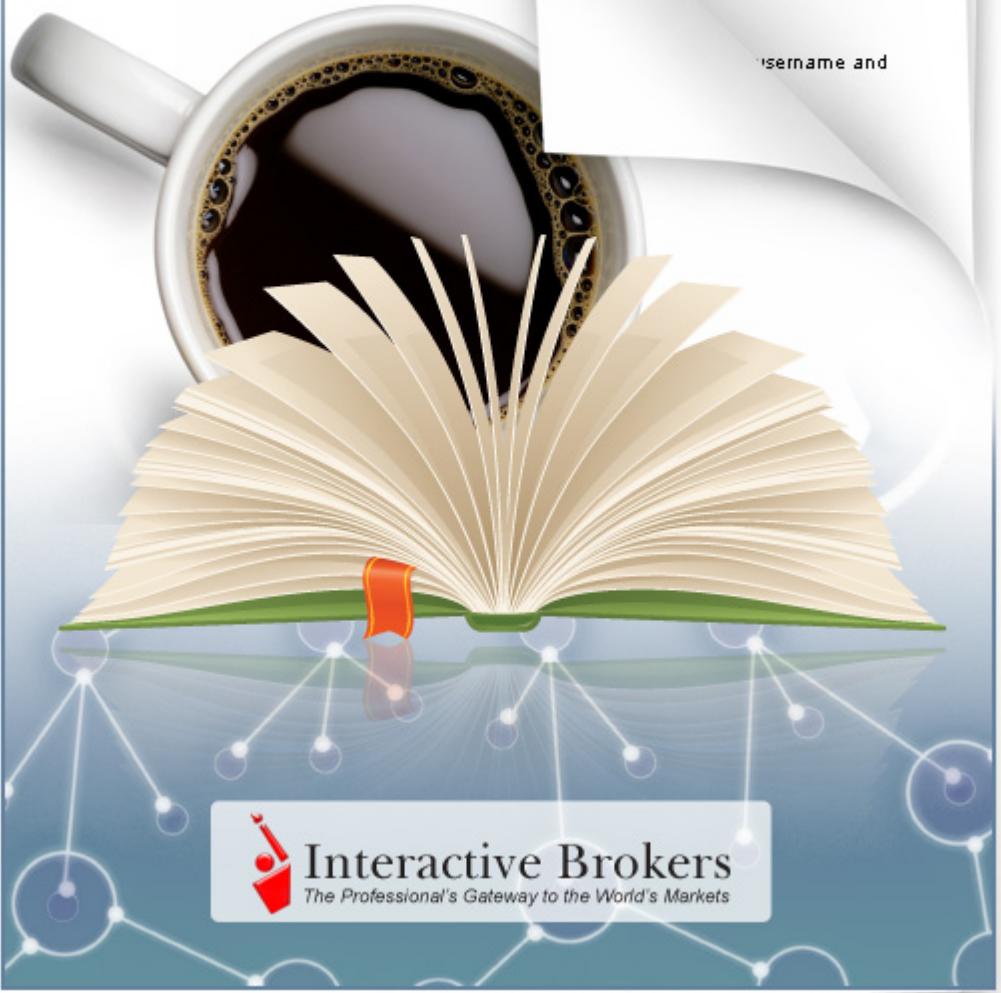


Getting Started with the TWS Java API

To download the software to your PC and
use it with your favorite Java IDE, you can use the
Java API. This version allows you to access your
TWS Java API directly from your Java application.
It is faster than the Java API and may run faster, but
it does not support all the new features. To download to
your PC, go to the Interactive Brokers website.

Select Trader Workstation

Enter your username and



Getting Started with the TWS Java API

March 2011

Supports TWS API Release 9.64

© 2011 Interactive Brokers LLC. All rights reserved.

Sun, Sun Microsystems, the Sun Logo and Java are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. Excel, Windows and Visual Basic (VB) are trademarks or registered trademarks of the Microsoft Corporation in the United States and/or in other countries.

Any symbols displayed within these pages are for illustrative purposes only, and are not intended to portray any recommendation.

Contents

1	Introduction	7
	How to Use this Book	8
	Organization	8
	Part 1: Introducing the TWS Java API	8
	Part 2: Preparing to Use the TWS Java API	8
	Part 3: Getting to Know the Java Test Client	9
	Part 4: Java Samples	9
	Part 5: Where to Go from Here.....	9
	Footnotes and References	9
	Icons	10
	Document Conventions.....	11
2	TWS and the Java API	13
	Chapter 1 - What is Trader Workstation?.....	14
	What Can You Do with TWS?	15
	A Quick Look at TWS.....	15
	The TWS Quote Monitor	15
	The Order Ticket	15
	Real-Time Account Monitoring	16
	Chapter 2 - Why Use the TWS Java API?.....	17
	TWS and the API	17
	Available API Technologies	18
	An Example	18
3	Preparing to Use the Java API.....	21
	Chapter 3 - Download the Java JDK and IDE.....	22
	Chapter 4- Download the API Software	23
	Chapter 5 - Connect to the Java Test Client.....	26
4	Market Data.....	31
	Chapter 6 - Connect the Java Test Client to TWS	32

Java Test Client Basic Framework	32
SampleFrame.java	33
What Happens When I Click the Connect Button?	34
Disconnecting from a Running Instance of TWS	37
Chapter 7: Requesting and Canceling Market Data	38
What Happens When I Click the Req Mkt Data Button?.....	39
The Sample Dialog	40
The reqMktData() Method.....	41
EWrapper Methods that Return Market Data.....	44
Getting a Snapshot of Market Data.....	45
Canceling Market Data.....	45
Chapter 8 - Requesting and Canceling Market Depth	46
What Happens When I Click the Req Mkt Depth Button?.....	47
The reqMktDepth() Method.....	47
The updateMktDepth() and updateMktDepthL2() Methods	48
Canceling Market Depth.....	48
Chapter 9 - Requesting and Canceling Historical Data	49
What Happens When I Click the Historical Data Button?	50
The reqHistoricalData() Method	50
The historicalData() Method	51
Canceling Historical Data	51
Chapter 10 - Requesting and Canceling Real Time Bars.....	52
What Happens When I Click the Req Real Time Bars Button?	53
The reqRealTimeBars() Method	53
The realtimeBar() Method.....	54
Canceling Real Time Bars.....	54
Chapter 11 - Subscribing to and Canceling Market Scanner Subscriptions	55
What Happens When I Click the Market Scanner Button?.....	56
The reqScannerParameters() and reqScannerSubscription() Methods	57
The scannerData() Method	57
The scannerDataEnd() Method	57
Cancel a Market Scanner Subscription	58
Chapter 12: Requesting Contract Data.....	59
What Happens When I Click the Req Contract Data Button?	59
The reqContractDetails() Method	60
The contractDetails() Method	60

5 Orders and Executions.....	61
Chapter 13: Placing and Canceling an Order	62
What Happens When I Place an Order?	63
The placeOrder() Method.....	65
The orderStatus() Method	66
Canceling an Order	66
Modifying an Order	66
Requesting "What-If" Data before You Place an Order.....	67
Chapter 14: Exercising Options.....	68
What Happens When I Click the Exercise Options Button?	68
The exerciseOptions() Method	70
Chapter 15: Extended Order Attributes	71
What Happens When I Click the Extended Button?	71
Chapter 16: Requesting Open Orders	73
Running Multiple API Sessions	73
The Difference between the Three Request Open Orders Buttons.....	74
What Happens When I Click the Req Open Orders Button?.....	74
The reqOpenOrders() Method.....	74
What Happens When I Click the Req All Open Orders Button?	75
The reqAllOpenOrders() Method	75
What Happens When I Click the Req Auto Open Orders Button?	76
The reqAutoOpenOrders() Method	76
Chapter 17 Requesting Executions	77
What Happens When I Click the Req Executions Button?	77
The reqExecutions() Method	78
The execDetails() Method	78
6 Additional Tasks	79
Chapter 18 - Requesting the Current Time	80
What Happens When I Click the Req Current Time Button?	80
Chapter 19: Subscribing to News Bulletins	81
What Happens When I Click the Req News Bulletins Button?	81
The reqNewsBulletins() method.....	82
The updateNewsBulletin() Method	82
Canceling News Bulletins	83
Chapter 20: View and Change the Server Logging Level.....	84

What Happens When I Click the Server Logging Button?	84
The setServerLogLevel() Method	85
7 Sample Applications for the Java API	87
Chapter 21 - Downloading and Preparing the Sample Code.....	88
Download the Samples	88
What's In the Zipped Sample File?	88
Setting Up the Project in NetBeans.....	89
A Quick Look at the New Project.....	91
Chapter 22 - Example 1: Requesting Market Data	92
Run Example 1	92
What Happens When You Run Example 1?	93
Looking at Example1.java	94
Connecting to TWS	96
Creating a Contract.....	97
Getting a Snapshot of Market Data.....	99
The while Loop	99
Getting the Last Price	100
Disconnecting from TWS	101
The build.xml Build File.....	101
Chapter 23 - Example 2: Automating Option Orders	102
Run Example 2	102
What Happens When You Run Example 2?	103
Looking at Example2.java	105
Connecting to TWS	109
Retrieving the Underlying Data	110
Creating a Contract	112
Requesting Market Data	113
The while Loop.....	114
Getting the Last Price, Option Implied Volatility and Historical Volatility	115
Retrieving Options Contracts	116
contractDetails() and contractDetailsEnd()	117
Placing the Straddle Order	119
Disconnecting from TWS	119
The build.xml Build File.....	120

8 Where to Go from Here.....	121
Chapter 24 - Linking to TWS using the TWS Java API.....	122
Chapter 25 - Additional Resources	125
Help with Java Programming	125
Help with the Java API.....	125
The API Reference Guide	125
The API Beta and API Production Release Notes.....	125
The TWS API Webinars.....	126
API Customer Forums	126
IB Customer Service	126
IB Features Poll.....	126
A Appendix A - Extended Order Attributes.....	127
B Appendix B - Account Page Values	131

Introduction

You might be looking at this book for any number of reasons, including:

- You love IB's TWS, and are interested in seeing how using its API can enhance your trading.
- You use another online trading application that doesn't provide the functionality of TWS, and you want to find out more about TWS and its API capabilities.
- You never suspected that there was a link between the worlds of trading/financial management and computer programming, and the hint of that possibility has piqued your interest.

Or more likely you have a reason of your own. Regardless of your original motivation, you now hold in your hands a unique and potentially priceless tome of information. Well, maybe that's a tiny bit of an exaggeration. However, the information in this book, which will teach you how to access and manage the robust functionality of IB's Trader Workstation through our TWS Java API, could open up a whole new world of possibilities and completely change the way you manage your trading environment. Keep reading to find out how easy it can be to build your own customized trading application.



If you are a Financial Advisor who trades for and allocates shares among multiple client accounts and would like more information about using the Java API, see the Getting Started with the TWS Java API for Advisors Guide.

How to Use this Book

Before you get started, you should read this section to learn how this book is organized, and see which graphical conventions are used throughout.

Our main goal is to give active traders and investors the tools they need to successfully implement a custom trading application (i.e. a trading system that you can customize to meet your specific needs), and that doesn't have to be monitored every second of the day. If you're not a trader or investor you probably won't have much use for this book, but please, feel free to read on anyway!



Throughout this book, we use the acronym "TWS" in place of "Trader Workstation." So when you see "TWS" anywhere, you'll know we're talking about Trader Workstation.



Before you read any further, we need to tell you that this book focuses on the TWS side of the Java API - we don't really help you to learn Java. If you aren't a fairly proficient Java programmer, or at least a very confident and bold beginner, this may be more than you want to take on. We suggest you start with a beginner's Java programming book, and come back to us when you're comfortable with Java.

Organization

We've divided this book into five major sections, each of which comprises a number of smaller subsections, and each of **those** have even smaller groupings of paragraphs and figures...well, you get the picture. Here's how we've broken things down:

Part 1: Introducing the TWS Java API

The chapters in this section help you answer those important questions you need to ask before you can proceed - questions such as "What can TWS do for me?" and "Why would I use an API?" and "If I WERE to use an API, what does the Java platform have to offer me?" and even "What other API choices do I have?"

If you already know you want to learn about the TWS API, just skip on ahead.

Part 2: Preparing to Use the TWS Java API

Part 2 walks you through the different things you'll need to do before your API application can effectively communicate with TWS. We'll help you download and install the API software, configure TWS and get the Java Test Client sample application up and running. A lot of this information is very important when you first get started, but once it's done, well, it's done, and you most likely won't need much from this section once you've completed it.

Part 3: Getting to Know the Java Test Client

Part 3 gets you working with the Java Test Client: learning how to request, receive and cancel market data, market depth, historical data, how to place an order, view execution reports, and monitor your account activity. We'll tell you exactly what methods you need to use to send info to TWS, and just what TWS will send you back. We've already documented the method parameters, descriptions and valid values in the API Reference Guide, so instead of duplicating efforts and filling this book up with those important reference tidbits, we provide targeted links to different sections of the users' guide as we need them.

Part 4: Java Samples

OK, here we're leaving the world of the known and venturing into new territory (which as everyone knows is actually *the* most exciting place to be). Now that you're familiar with our stuff and how it works, it's time to leave the nest and go out on your own. This section helps you to get started using our TWS Java API to create an application that does what YOU want by introducing you to two custom-written programs. Of course to protect your million-dollar inspiration, you'll have to implement your ideas on your own.

Part 5: Where to Go from Here

After filling your head with boatfuls of API knowledge, we wouldn't dream of sending you off empty-handed! Part 5 includes some additional information about linking to TWS using our Java API, then tells you how to keep abreast of new API releases (which of course means new features you can incorporate into your trading plan), how to navigate the Interactive Brokers website to find support and information, and what resources we recommend to help you answer questions outside the realm of IB support, questions such as "Why isn't my Java JDK working?"

Footnotes and References

¹Any symbols displayed are for illustrative purposes only and are not intended to portray a recommendation.

Icons



TWS-Related

When you see this guy, you know that there is something that relates specifically to TWS: a new feature to watch for, or maybe something you're familiar with in TWS and are looking for in the API.



Java Tip

The Java tips are things we noted and think you might find useful. They don't necessarily relate only to TWS. We don't include too many of these, but when you see it you should check it out - it will probably save you some time.



Important!



You may want to take a peek, but it isn't the end of the world if you don't.

Take a Peek!



This icon denotes references outside of this book that we think may help you with the current topic, including links to the internet or IB site, or a book title.

Go Outside!

Document Conventions

Here's a list of document conventions used in the text throughout this book.

Convention	Description	Examples
Bold	Indicates: <ul style="list-style-type: none">• menus• screens• windows• dialogs• buttons• tabs• keys you press• names of classes and methods	On the Tickers page, select a row by clicking the row number in the far left column... Press Ctrl+C to copy...
<i>Italics</i>	Indicates: <ul style="list-style-type: none">• commands in a menu• objects on the screen, such as text fields, check boxes, and drop-down lists	To access the users' guide, under the Software menu, select <i>Trader Workstation</i> , then click <i>Users' Guide</i> .

In addition, Java code snippets appear in the following format:

EClientSocket constructor

```
EClientSocket m_client = new EClientSocket(this);
```

Introduction

How to Use this Book

TWS and the Java API

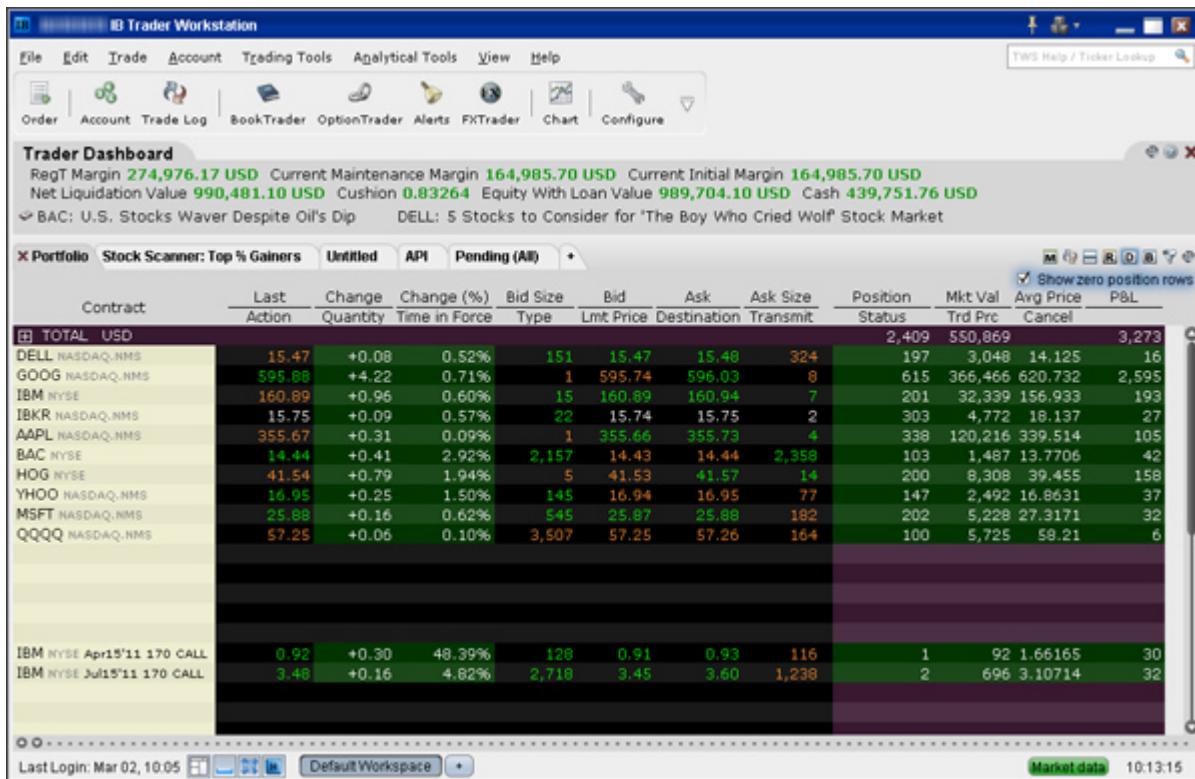
The best place to start is by getting an idea of what Trader Workstation (TWS), is all about. In this section, first we'll describe TWS and some of its major features. Then we'll explain how the API can be used to enhance and customize your trading environment. Finally, we'll give you a summary of some of the things the Java API can do for you!

Here's what you'll find in this section:

- [Chapter 1 - What is Trader Workstation?](#)
- [Chapter 2 - Why Use the TWS Java API?](#)

Chapter 1 - What is Trader Workstation?

Interactive Brokers' Trader Workstation, or TWS, is an online trading platform that lets you trade and manage orders for all types of financial products (including stocks, bonds, options, futures and Forex) on markets all over the world - all from a single spreadsheet-like screen.



To get a little bit of a feel for TWS, go to the IB website and try the TWS demo application. Its functionality is slightly limited and it only supports a small number of symbols, but you'll definitely get the idea. Once you have an approved, funded account you'll also be able to use PaperTrader, our simulated trading tool, with paper-money funding in the amount of \$100,000, which you can replenish at any time through TWS Account Management.

What Can You Do with TWS?

So, what can you do with TWS? For starters, you can:

- Send and manage orders for all sorts of products (all from the same screen!);
- Monitor the market through Level II, NYSE Deep Book and IB's Market Depth;
- Keep a close eye on all aspects of your account and executions;
- Use Technical, Fundamental and Price/Risk analytics tools to spot trends and analyze market movement;
- Completely customize your trading environment through your choice of modules, features, tools, fonts and colors, and user-designed workspaces.

Basically, almost anything you can think of TWS can do - or will be able to do soon. We are continually adding new features, and use the latest technology to make things faster, easier and more efficient. As a matter of fact, it was this faith in technology's ability to improve a trader's success in the markets (held by IB's founder and CEO Thomas Peterffy) that launched this successful endeavor in the first place. Since the introduction of TWS in 1995, IB has nurtured this relationship between technology and trading almost to the point of obsession!

A Quick Look at TWS

This section gives you a brief overview of the most important parts of TWS.

The TWS Quote Monitor

First is the basic TWS Quote Monitor. It's laid out like a spreadsheet with rows and columns. To add tickers to a page, you just click in the Underlying column, type in an underlying symbol and press Enter, and walk through the steps to select a product type and define the contract. Voila! You now have a live market data line on your trading window. It might be for a stock, option, futures or bond contract. You can add as many of these as you want, and you can create another window, or trading page, and put some more on that page. You can have any and all product types on a single page, maybe sorted by exchange, or you can have a page for stocks, a page for options, etc. Once you get some market data lines on a trading page, you're ready to send an order.

The Order Ticket

What? An order ticket? Sure, we have an order ticket if that's what you really want. But we thought you might find it easier to simply click on the bid or ask price and have us create a complete order line instantly, right in front of your eyes! Look it over, and if it's what you want click a button to transmit the order. You can easily change any of the order parameters right on the order line. Then just click the green Transmit guy to transmit your order! It's fast and it's easy, and you can even customize this minimal two-click procedure (by creating hotkeys and setting order defaults for example) so that you're creating and transmitting orders with just ONE click of the mouse.

Real-Time Account Monitoring

TWS also provides a host of real-time account and execution reporting tools. You can go to the Account Window at any time to see your account balance, total available funds, net liquidation and equity with loan value and more. You can also monitor this data directly from your trading window using the Trader Dashboard, a monitoring tool you can configure to display the last price for any contracts and account-related information directly on your trading window.

So - TWS is an all-inclusive, awesome powerful trading tool. You may be wondering, "Where does an API fit in with this?" Read on to discover the answer to that question.



For more information on TWS, see the TWS Users' Guide on our web site.

Chapter 2 - Why Use the TWS Java API?

OK! Now that you are familiar with TWS and what it can do, we can move on to the amazing API. If you actually read the last chapter, you might be thinking to yourself "Why would I want to use an API when TWS seems to do everything." Or you could be thinking "Hmmmm, I wonder if TWS can... fill in the blank?" OK, if you're asking the first question, I'll explain why you might need the API, and if you're asking the second, it's actually the API that can fill in the blank.

TWS has the capability to do tons of different things, but it does them in a certain way and displays results in a certain way. It's likely that our development team, as fantastic as they are, hasn't yet exhausted the number of features and way of implementing them that all of you collectively can devise. So it's very likely that you, with your unique way of thinking, will be or have been inspired by the power of TWS to say something like "Holy moly, I can't believe I can really do all of this with TWS! Now if I could only just (fill in the blank), my life would be complete!"

That's where the API comes in. Now, you can fill in the blank! It's going to take a little work to get there, but once you see how cool it is to be able to access functionality from one application to another, you'll be hooked.

TWS and the API

In addition to allowing you pretty much free reign to create new things and piece together existing things in new ways, the API is also a great way to automate your tasks. You use the API to harness the power behind TWS - in different ways.

Here's an analogy that might help you understand the relationship between TWS and the API. Start by imagining TWS as a book (since TWS is constantly being enhanced, our analogy imagines a static snapshot of TWS at a specific point in time). It's the reference book you were looking for, filled with interesting and useful information, a book with a beginning, middle and end, which follows a certain train of logic. You could skip certain chapters, read Chapter 10 first and Chapter 2 last, but it's still a book. Now imagine, in comparison, that the API is the word processing program in which the book was created with the text of the book right there. This allows you access to everything in the book, and most importantly, it lets you continually change and update material, and automate any tasks that you'd have to perform manually using just a book, like finding an index reference or going to a specific page from the table of contents.

The API works in conjunction with TWS and with the processing functions that run behind TWS, including IB's SmartRouting, high-speed order transmission and execution, support for over 40 orders types, etc. TWS accesses this functionality in a certain way, and you can design your API to take advantage of it in other ways.

Available API Technologies

IB provides a suite of custom APIs in multiple programming languages, all to the same end. These include Java, C++, Active X for Visual Basic and .NET, and DDE for Excel (Visual Basic for Applications, or VBA). This book focuses specifically on just one, the Java version. Why would you use Java over the other API technologies? The main reason might be that you are a Java expert. If you don't know Java or any other programming language, you should take a look at the Excel/DDE API, which has a much smaller learning curve. But if you know Java, this platform offers more flexibility than the DDE for Excel, is supported on Windows, MAC, and Unix/Linux (the DDE is only supported in Windows), and provides very high performance.

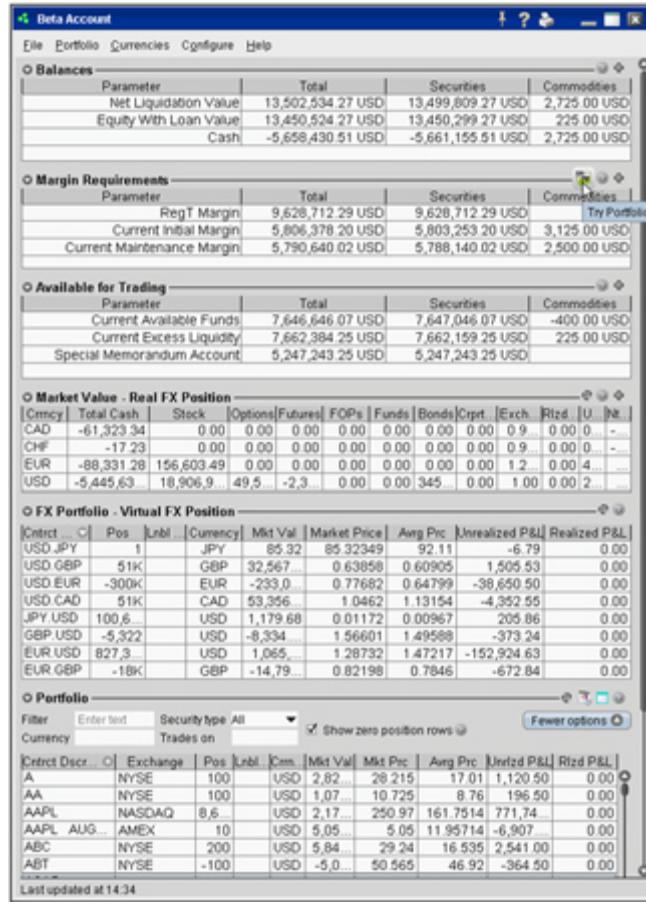


For more information about our APIs, see the Application Programming Interfaces page on our web site.

An Example

It's always easier to understand something when you have a real life example to contemplate. What follows is a simple situation in which the API could be used to create a custom result.

TWS provides an optional field that shows you your position-specific P&L for the day as either a percentage or an absolute value. Suppose you want to modify your position based on your P&L value? At this writing, the only way to do this would be to watch the market data line to see if the P&L changed, and then manually create and transmit an order, but only if you happened to catch the value at the right point. Hmmmm, I don't think so! Now, enter the API! You can instruct the API to automatically trigger an order with specific parameters (such as limit price and quantity) when the P&L hits a certain point. Now that's power! Another nice benefit of the API is that it gives you the ability to use the data in TWS in different ways. We know that TWS provides an extensive Account Information window that's chock-full of everything you'll ever want to know about your account status. The thing is, it's only displayed in a TWS window, like this:



Lovely though it is, what if you wanted to do something else with this information? What if you want it reflected in some kind of banking spreadsheet where you log information for all accounts that you own, including your checking account, Interactive Brokers' account, 401K, ROIs, etc? Again - enter the API!

You can instruct the API to get any specific account information and put it wherever it belongs in a spreadsheet. The information is linked to TWS, so it's easy to keep the information updated by simply linking to a running version of TWS. With a little experimenting, and some help from the *API Reference Guide* and the *TWS Users' Guide*, you'll be slinging data like a short-order API chef in no time!

There are a few other things you must do before you can start working with the TWS Java API. The next chapter gets you geared up and ready to go.

Preparing to Use the Java API

Although the API provides great flexibility in implementing your automated trading ideas, all of its functionality runs through TWS. This means that you must have a TWS account with IB, and that you must have your TWS running in order for the API to work. This section takes you through the minor prep work you will need to complete, step by step.

Here's what you'll find in this section:

- [Chapter 3 - Download the Java JDK and IDE](#)
- [Chapter 4- Download the API Software](#)
- [Chapter 5 - Connect to the Java Test Client](#)



We want to tell you again that this book focuses on the TWS side of the Java API - we don't really help you to learn Java. Unless you are a fairly proficient Java programmer, or at least a very confident and bold beginner, this may be more than you want to take on. We suggest you start with a beginner's Java programming book, and come back to us when you're comfortable with Java.

Chapter 3 - Download the Java JDK and IDE

OK, well we've already said that you need to know Java before you can successfully implement your own TWS Java API application, and there's a good chance you already have the Java tools you'll need downloaded and installed. But in case you don't, we'll quickly walk you through what you need:

- The Java development kit (JDK)
- An integrated development environment (IDE).

We like the J2SE Development Kit and NetBeans IDE Bundle that's available (free!) from the Sun website. We're not including any version numbers of these Sun Java products, as they'll likely be different by the time you read this. You can use any IDE you're comfortable with.



In this book we use NetBeans as the IDE of choice, so if you're using another IDE you'll have to reinterpret our instructions to fit your development environment. If you're using NetBeans and aren't totally familiar with it, we recommend browsing through the Quick Start or the tutorial, both of which are available on the Help menu.

Anyway, I know we're not giving you too much here, but we are assuming you have enough savvy to find this stuff, download it, and install it. This is a tough line for us to walk, because we're really focusing on the TWS Java API for beginners, not on Java for beginners. If you're having trouble at this point, you should probably start with the TWS DDE for Excel API to get your feet wet!

Once you have these pieces downloaded and installed, you can go to the IB website and download the TWS API software.

Chapter 4- Download the API Software

Next, you need to download the API software from the IB website.

Step 1: Download the API software.

This step takes you out to the IB website at

http://individuals.interactivebrokers.com/en/p.php?f=programInterface&p=a&ib_entity=lic.

The menus are along the top of the homepage. Hold your mouse pointer over the Trading menu, then click *API Solutions*.



On the API Solutions page, click the **IB API** button on the left side of the page.



This displays the IB API page which shows a table with links to software downloads that are compatible with Windows, MAC or Unix platforms. When available, there will also be a Windows Beta version of the software. Look across the top of the table and find the OS you need.

IB API

PDF Print Friendly

Software Connectivity Getting Started Guides Reference Guide Release Notes OCC Option Symbology

IB API Software

Program traders may build their own add-on applications in Excel (using DDE or ActiveX), C++, Posix C++, Java, and Visual Basic with our proprietary IB Application Program Interface (API), which requires [connectivity](#) via either the TWS or the IB Gateway. We encourage API users to test their API components with their PaperTrader or the [TWS Demo System](#) before actually implementing any new API systems.

	Windows	Windows Beta	MAC/UNIX		MAC/UNIX Beta	
Software	Download latest version Downgrade to previous version	Download beta version	Download and Installation Instructions for MAC	Download and Installation Instructions for UNIX	Download and Installation Instructions for MAC	Download and Installation Instructions for UNIX
Released Date	May 12, 2009	May 05, 2010	May 11, 2009		May 05, 2010	
Version	API 9.63	API beta 9.64	API 9.63		API beta 9.64	
Special Notes	Includes the C++ Socket, Java Socket, DDE, Active X APIs, and sample code for each.			Includes the Java Socket API, Posix C++ Socket API and sample code for each.		
Support	API Reference Guide or IB Discussion Forum .					

As a reminder, the use of the IB API as a means of disseminating information, including market data or any other licensed or copyrighted information, to third parties or non-registered IB customers is strictly prohibited without prior written approval of Interactive Brokers.



For this book, we assume that you are using Windows. If you're using a different operating system (Mac, Unix), be sure to adjust the instructions accordingly!

In the Windows column, click *Download Latest Version*. This opens a File Download box, where you can decide whether to save the installation file, or open it. We recommend you choose Save and then select a place where you can easily find it, like your desktop (you choose the path in the Save in field at the top of the Save As box that opens up). Once you've selected a good place to put it, click the **Save** button. It takes seconds to download the executable file. Note that the API installation file is named for the API version; for example, InstallAX_960.



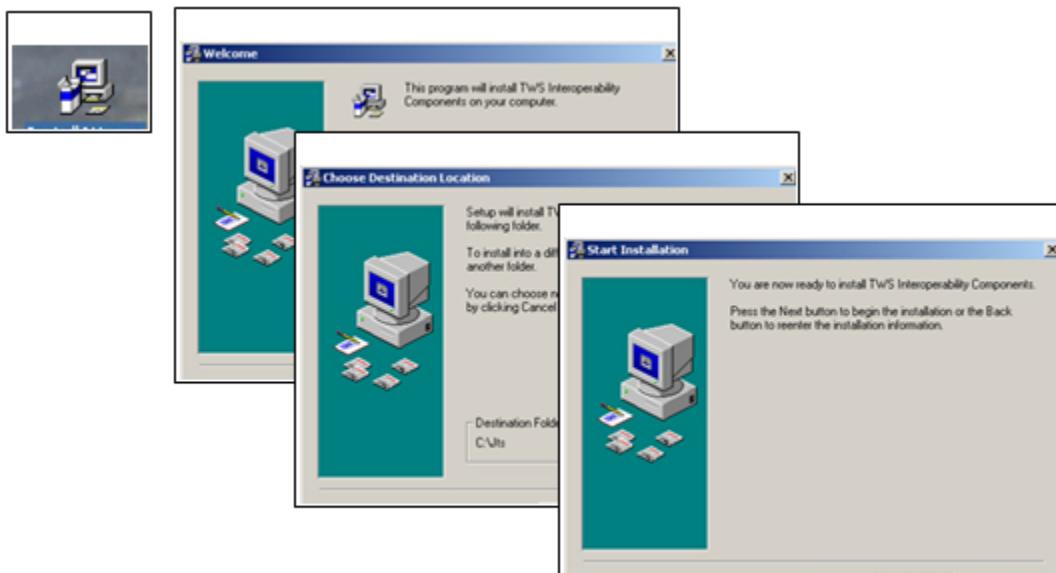
We'll usually be stressing just the opposite, but at this point, you need to make sure TWS is not running. If it is, you won't be able to install the API software.

Step 2: Install the API software.

Next, go to the place where you saved the file (for example, your desktop or some other location on your computer), and double-click the API software installation file icon. This starts the installation wizard, a simple process that displays a series of dialogs with questions that you must answer.



Remember where the installation wizard installs the application. You'll need this information later when you open the API application in Excel.



Once you have completed the installation wizard, the sample application installs, and you're ready to open the Java Test Client, connect to TWS, and get started using the Java API sample application!

Chapter 5 - Connect to the Java Test Client

OK, you've got all the pieces in place. Now that we're done with the prep work, it's time to get down to the fun stuff.

Although the API provides great flexibility in implementing your automated trading ideas, all of its functionality runs through TWS. This means that you must have a TWS account with IB, and you must have TWS running in order for the API to work. This section describes how to enable TWS to connect to the Java API. Note that if you don't have an account with IB, you can use the Demo TWS system to check things out.. If you DO have an account, we recommend opening a linked PaperTrader test account, which simulates the TWS trading environment, and gives you \$100,000 in phantom cash to play with.

Enabling TWS to support the API is probably the simplest step you'll encounter in this book. It's probably more difficult to actually remember to log into TWS before you run the API!

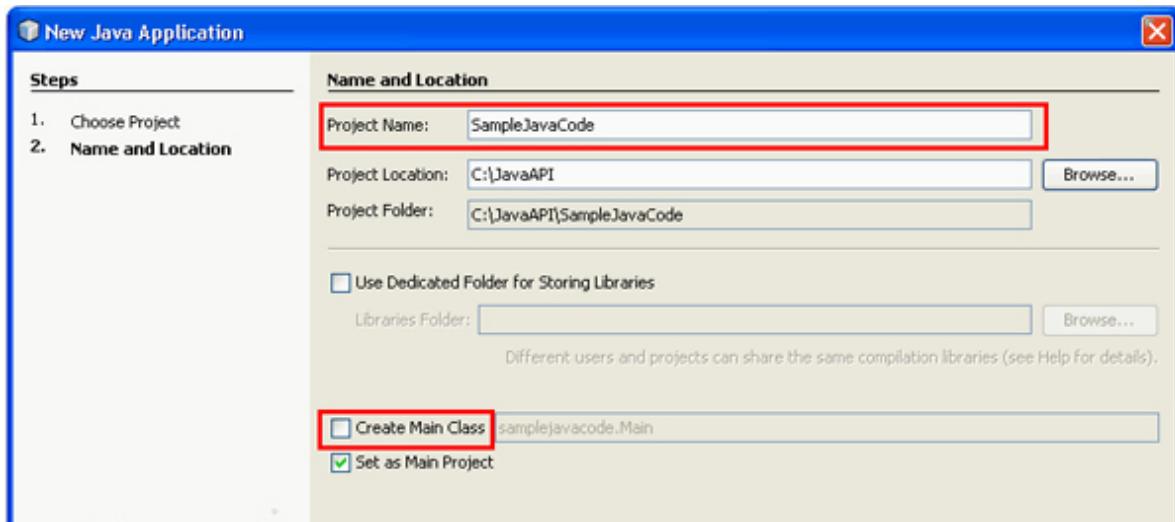
Step 1: Log into TWS.

OK, log into TWS, or run the Demo available on the **Demo** tab of the Trader Workstation page on our website.

Now look up at the top of the trading window, and you'll see the menu bar. Click the **Edit** menu, and then click *Global Configuration*. In the Configuration window, click *API* in the left pane, then click *Settings*, which reveals several options on the right side of the window. Check the *Enable ActiveX and Socket Clients* check box and click **OK**.

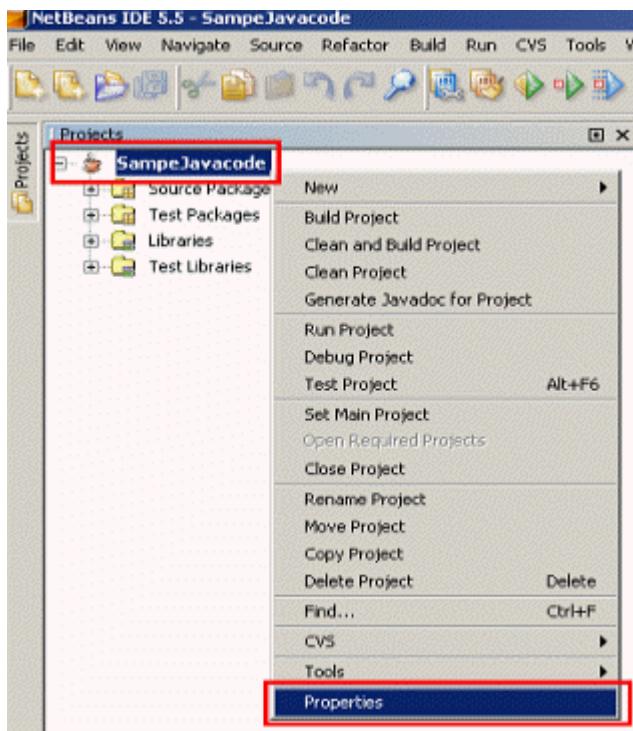
Step 2: Set Up the Java Test Client.

Now, open NetBeans and click *New Project*. This starts the project wizard. In the Projects area, select *Java Application* and click **Next**. You'll see a screen like the one below.

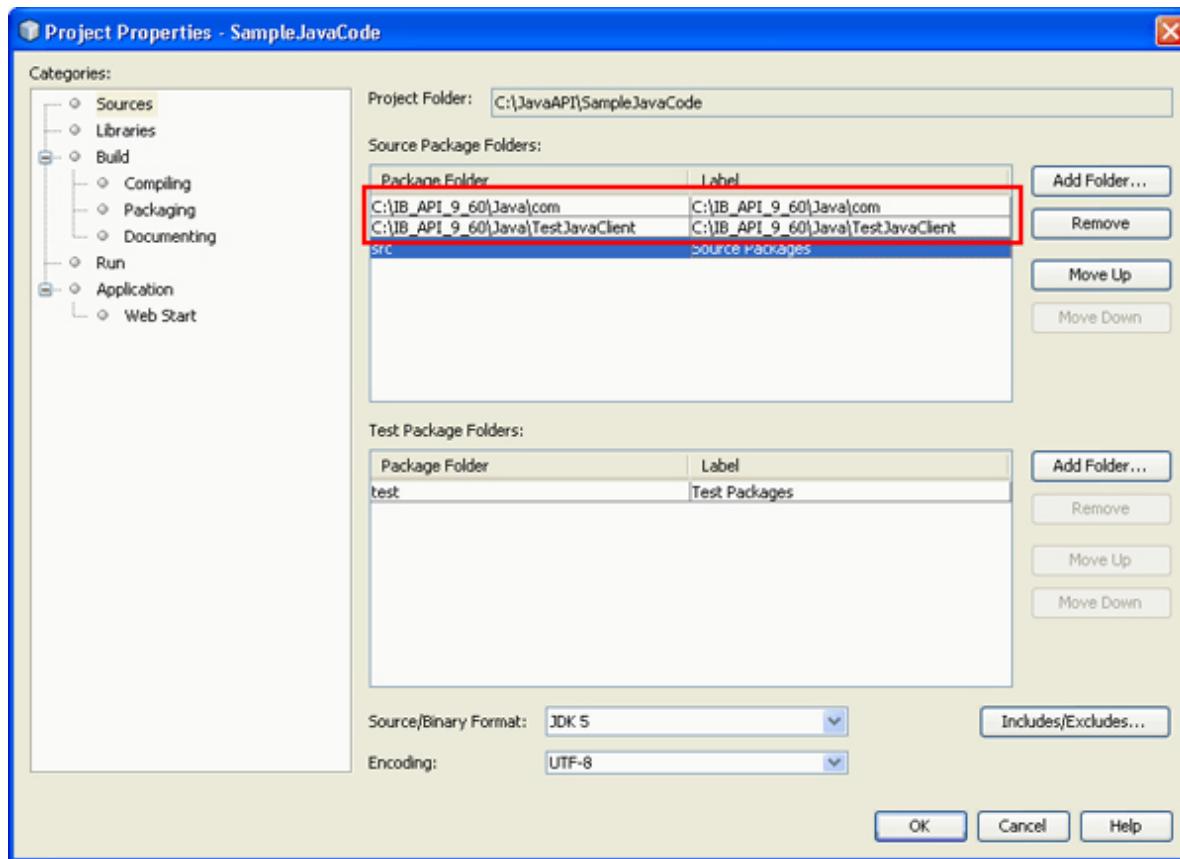


Enter a project name and project location. Uncheck the box for *Create Main Class* and click **Finish**.

Now right-click your new *SampleJavaCode* project from the *Projects* list and select *Properties*.



In the Source Package Folders area, click **Add Folder** and navigate to the directory where you installed the API sample program. Add two folders: the com folder and the *TestJavaClient* folder. Then click **OK**.



Step 3: Run the Java Test Client.

Now it's time to run the application. Press **F6** to run. When the system prompts you to select *TestJavaClient.Main* as the main class, click **OK** (recall that earlier, you had to uncheck the *Create Main Class* box when you first set up the project; now is when you assign the main class). And of course, click **OK** again.

Now press **F6** to run again. You're looking at the java test client, and you should see something like this thing below:



Here you are. What now? Part II focuses on performing the trading tasks defined by the action buttons in the sample client. We'll take a quick, general look at what's going on behind the GUI. Then we'll walk through the basics of the TWS API, in the order defined by the buttons in the Java Test Client layout, pictures above.



The TWS API does not have to be written as a GUI program, but to completely understand how the Java Test Client works, you should have some general understanding of Java Swing. We recommend taking a look at The Swing Tutorial on java.sun.com.

Market Data

You've completed the prep work, and you have the Java Test Client up and running. This section of the book starts with a description of the basic framework of the Java Test Client, then reviews the TWS Java API methods associated with each trading task.

In the following chapters, we'll show you the methods and parameters behind this sample application, and how they call the methods and parameters in the TWS Java API.

Here's what you'll find in this section:

- [Chapter 6 - Connect the Java Test Client to TWS](#)
- [Chapter 7: Requesting and Canceling Market Data](#)
- [Chapter 8 - Requesting and Canceling Market Depth](#)
- [Chapter 9 - Requesting and Canceling Historical Data](#)
- [Chapter 10 - Requesting and Canceling Real Time Bars](#)
- [Chapter 11 - Subscribing to and Canceling Market Scanner Subscriptions](#)
- [Chapter 12: Requesting Contract Data](#)

Using the Java Test Client is a good way to practice locating and using the reference information in the API Reference Guide. With the sample program, you can compare the data in the sample message with the method parameters in the API Reference Guide.

Chapter 6 - Connect the Java Test Client to TWS

This chapter describes the basic framework of the Java Test Client and what happens when you connect and disconnect to a running instance of TWS.

Java Test Client Basic Framework

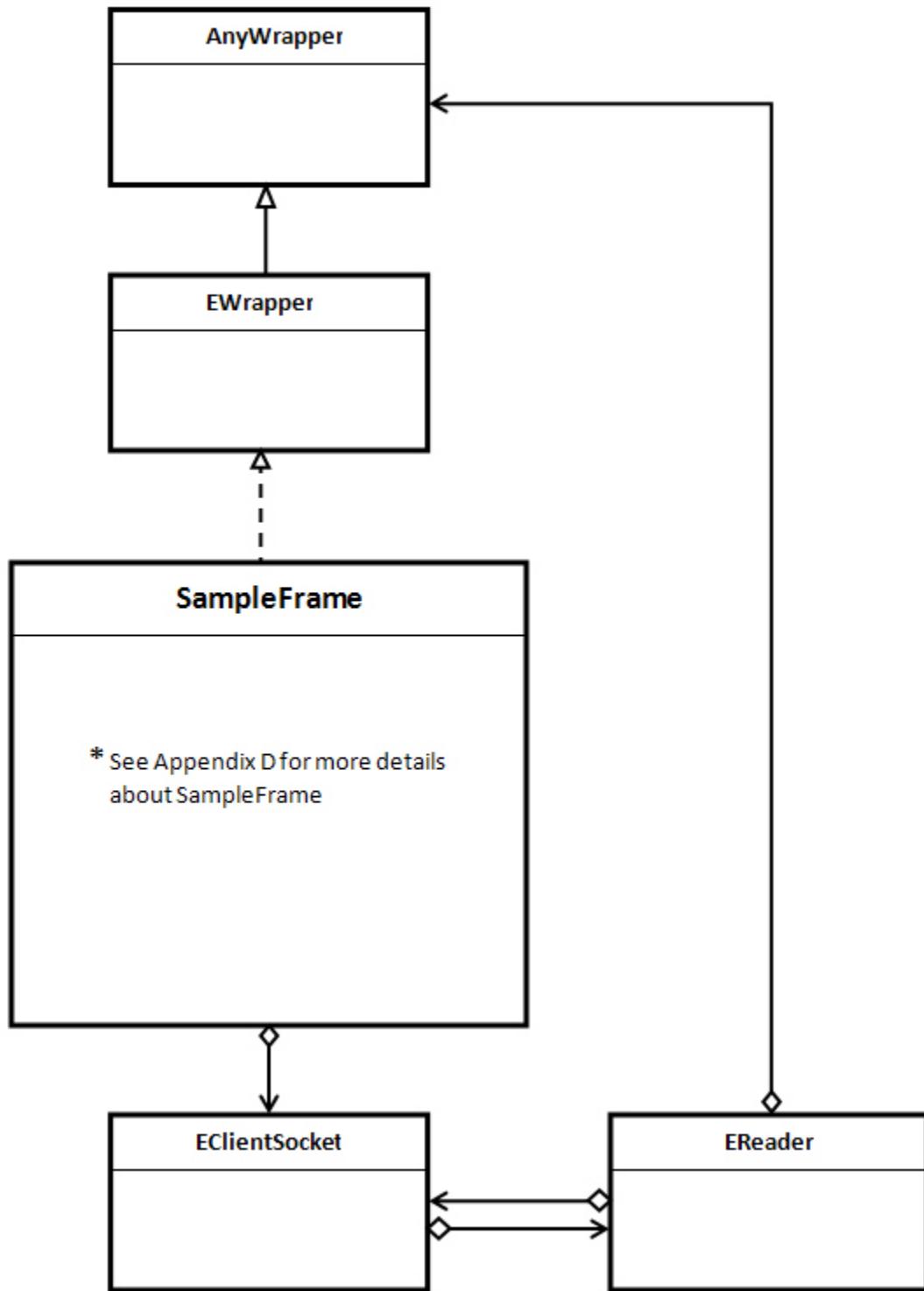
Let's take a look at the basic framework of the Java Test Client and the Java API. Here's the Java Test Client when you first run it:



The black text panels and the buttons that you see in the Java Test client are defined in the **SampleFrame.java** class, which is described on the next page.

SampleFrame.java

Here is a simplified class diagram that shows how SampleFrame.java in the Java Test Client uses our TWS Java API.



The most important things to remember here are that the SampleFrame.java class implements the **EWrapper** interface, which is the part of our TWS Java API that defines the methods that receive messages from TWS, and calls the methods in **EClientSocket**, which are used to send messages to TWS.



Throughout this book, we've included links to related help topics in the online API Reference Guide. So if you see a link, feel free to click it if you want to learn more about a particular class or method in our TWS Java API

Class definition of SampleFrame.java

```
class SampleFrame extends JFrame implements EWrapper
```

The SampleFrame.java class is where we attach an EClientSocket. The **EClientSocket** class is used to send messages to TWS. We will call methods such as eConnect(), reqMarketData(), etc. on the EClientSocket object. In the example below, we are passing **this** object to the constructor of EclientSocket - that is, the current instance of the SampleFrame class (which is an instance of EWrapper).

EClientSocket constructor

```
EClientSocket m_client = new EClientSocket(this);
```

What Happens When I Click the Connect Button?

The SampleFrame.java class also contains the **createButtonPanel()** method, which is where all the buttons on the Java Test client are defined.



The SampleFrame class and the createButtonPanel() method are unique to the sample application; they are not part of the TWS Java API and therefore are not documented in the API Reference Guide.

createButtonPanel() method

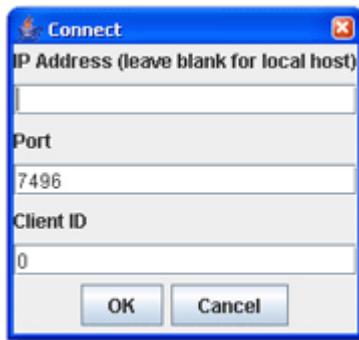
```
private JPanel createButtonPanel() {
    JPanel buttonPanel = new JPanel( new GridLayout( 0, 1 ) );
    .
    .
    .
}
```

Each button in the Java Test Client is really just a façade - a front for an *ActionListener*, which defines the method to be called when the button is clicked. The *ActionListener* for the **Connect** button is shown below.

ActionListener for the Connect button in createButtonPanel()

```
JButton butConnect = new JButton( "Connect");
butConnect.addActionListener( new ActionListener() {
    public void actionPerformed( ActionEvent e) {
        onConnect();
    }
});
```

So, when you click the **Connect** button, the **onConnect()** method of the *SampleFrame* class executes. A *ConnectDlg* object, an extension of *JDialog*, is instantiated. When the *setVisible(true)* method is called, the dialog is displayed.



Here is what the complete **onConnect()** method looks like.

onConnect() method

```
void onConnect() {
    m_bIsFAAccount = false;
    // get connection parameters
    ConnectDlg dlg = new ConnectDlg(this);
    dlg.setVisible(true);
    if( !dlg.m_rc) {
        return;
    }

    // connect to TWS
    m_disconnectInProgress = false;

    m_client.eConnect( dlg.m_retIpAddress, dlg.m_retPort,
                       dlg.m_retClientId);
    if (m_client.isConnected()) {
        m_TWS.add("Connected to Tws server version " +
                  m_client.serverVersion() + " at " +
                  m_client.TwsConnectionTime());
    }
}
```

And then you can enter the IP Address, Port and Client Id values in the input fields of the dialog. When you click the **OK** button (which has its own ActionListener defined in the ConnectDlg class), the onOk() method is executed.

The input values are retrieved and stored in the ConnectDlg object. A return code is set to true indicating success and the dialog is closed by calling the setVisible(false) method. In the user interface, a confirmation dialog is displayed.



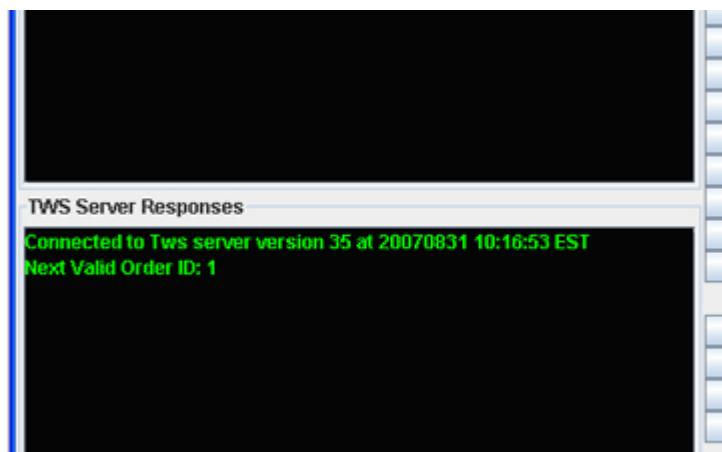
Control is now passed back to the **onConnect()** method in the SampleFrame. The return code was true, so processing continues. As shown in the following code, which is a portion of the **onConnect()** method, we make a call to the **eConnect()** method of the **EClientSocket** using the attributes of the ConnectDlg object as the parameters for IP Address, Port and Client Id. We can then check if the connection was successful by calling the **isConnected()** method.

In the code below, we add a message to the *m_TWS* object, which is an extension of the JPanel class. The end result is the message *Connected to Tws server version...* which displays in the *TWS Server Responses* panel of the Java Test Client.

Connecting to TWS in the onConnect() method

```
m_client.eConnect( dlg.m_retIpAddress, dlg.m_retPort,
dlg.m_retClientId);
if (m_client.isConnected()) {
    m_TWS.add("Connected to Tws server version " +
m_client.serverVersion() + " at " +
m_client.TwsConnectionTime());
}
```

If the connection is successful, a message is displayed in the *TWS Server Responses* panel of the Java Test Client as shown in the following figure.



Disconnecting from a Running Instance of TWS

A screenshot of a Java Swing application window. At the top is a menu bar with 'File', 'Edit', 'View', 'Help', and 'About'. Below the menu is a toolbar with several icons. In the center is a panel containing a single button labeled 'Disconnect'.

When you click the **Disconnect** button, the ActionListener in SampleFrame.java calls the **onDisconnect()** method (also in SampleFrame.java), which in turn calls the **eDisconnect()** method in the Java API **EClientSocket** object.

ActionListener for the Disconnect button (defined in createButtonPanel() method)

```
 JButton butDisconnect = new JButton( "Disconnect") ;
butDisconnect.addActionListener( new ActionListener() {
    public void actionPerformed( ActionEvent e) {
        onDisconnect();
    }
});
```

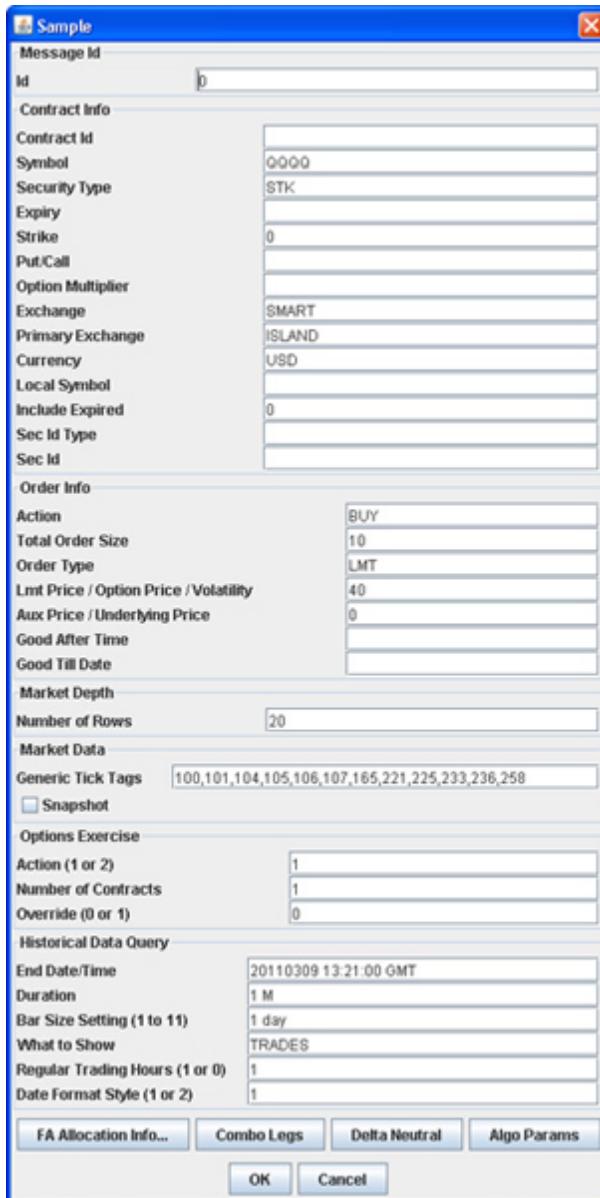
Disconnecting from TWS in the onDisconnect() method of SampleFrame.java

```
m_disconnectInProgress = true;
m_client.eDisconnect();
}
```

OK, got all of that? Great! Now let's move on, and see what happens when you use the market data buttons.

Chapter 7: Requesting and Canceling Market Data

This chapter describes how the Java Test Client requests and cancels market data. When you click one of the market data buttons, the Sample dialog appears.



This is the `orderDlg` object, an extension of `JDialog`, that is instantiated when one of the methods in the `ActionListeners` executes.

What Happens When I Click the Req Mkt Data Button?

Once you connect to TWS using the Java Test Client, you get market data by clicking the button, entering an underlying and some other information, and clicking **OK**. But what are you really doing? Each time you click a button in the sample client, an ActionListener with a defined method does its thing.



When you click the **Req Mkt Data** button, the attached ActionListener (shown below) "listens" to your request and puts the appropriate method, **onReqMktData()**, into "action!"

ActionListener for the Req Mkt Data button in createButtonPanel()

```
 JButton butMktData = new JButton ( "Req Mkt Data" );
butMktData.addActionListener ( new ActionListener() {
    public void actionPerformed ( ActionEvent e) {
        onReqMktData();
```

Next, the method defined in the ActionListener, **onReqMktData()**, opens the Sample dialog (i.e.instantiating the orderDlg object) so that you can fill in market data fields.

onReqMktData() method in SampleFrame.java

```
void onReqMktData() {
// run m_orderDlg
m_orderDlg.show();
if( !m_orderDlg.m_rc ) {
    return;
}
// req mkt data
m_client.reqMktData( m_orderDlg.m_id, m_orderDlg.m_contract,
    m_orderDlg.m_genericTicks, m_orderDlg.m_snapshotMktData);
}
```

Remember that the SampleFrame.java class is where we attach an EclientSocket, which is the object used to send messages to TWS.

When you click **OK**, the **reqMktData()** method sends your market data request to TWS and, if everything you entered is valid, the data you requested is returned by way of the **tickPrice()**, **tickSize()**, **tickGeneric()**, **tickOptionComputation()**, **tickString()** and **tickEFP()** methods.

When you use the *EClientSocket* object to send a message to TWS via a specific method (**reqMktData()**), you'll receive data back from TWS via a socket with implementations of specific methods (**tickPrice()**, **tickSize()**, etc.).



The methods called by the ActionListener are unique to the sample application; they are not part of the TWS Java API and therefore are not documented in the API Reference Guide.

The Sample Dialog

A sample dialog appears, divided into seven different areas of information. Each of these areas relates to a method parameter.

Note that this same dialog is displayed for many of the action buttons on the list. You need to know which methods/parameters (represented by fields in the Sample box) need data for each specific request.

The reqMktData() Method

Let's find out which parameters to use for requesting market data. The Class EClientSocket **reqMktData()** method looks like this:

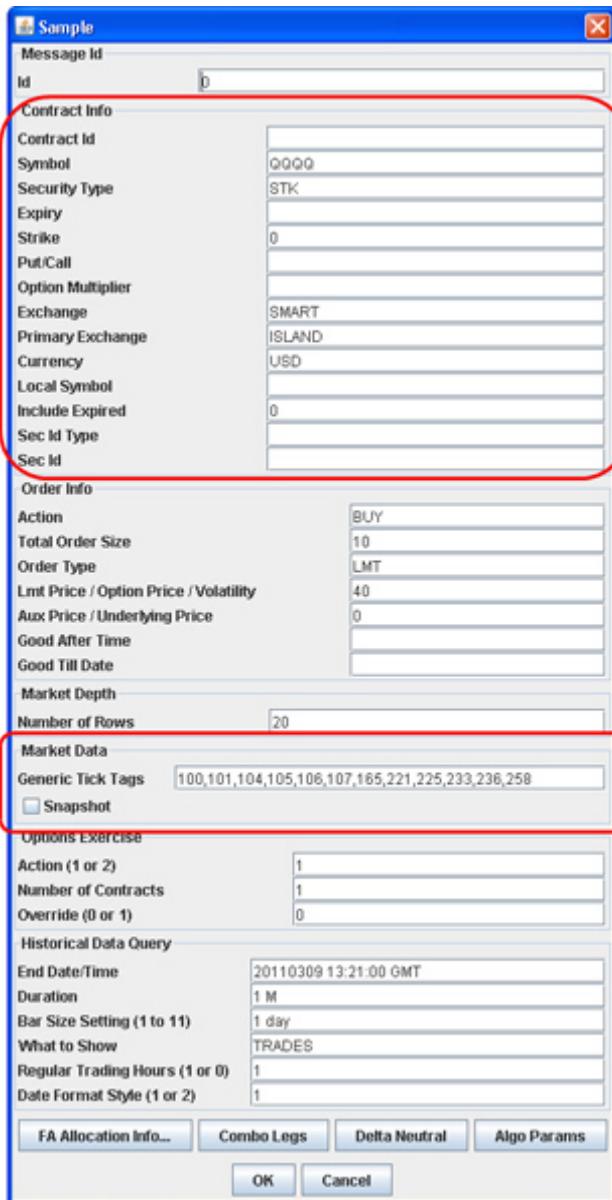
```
void reqMktData(int tickerId, Contract contract, String  
genericTicklist, boolean snapshot)
```

Parameter	Description
tickerId	The ticker id. Must be a unique value. When the market data returns, it will be identified by this tag. This is also used when canceling the market data.
contract	This class contains attributes used to describe the contract.
genericTicklist	A comma delimited list of generic tick types.
snapshot	Check to return a single snapshot of market data and have the market data subscription cancel. Do not enter any genericTicklist values if you use snapshot.

This table is for illustrative purposes only and is not intended to portray valid API documentation.

As you can see from the table above, this method has four parameters, the first three of which correspond to the fields in the Order dialog that you fill in.

Now let's take a look at the sample entry box you got when you clicked the **Req Mkt Data** button and see how and where the two relate.



The circled sections in the picture above (*Message ID*, *Contract Info* and *Market Data*) correspond to the parameters in the **reqMktData()** method (*tickerID*, *contract*, *genericTickList*, and *snapshot*). The contract class contains the parameters that correspond to the fields in the *Message Id* and *Contract Info* sections of the Sample dialog. You can ignore the other fields in the Sample dialog right now, because they represent parameters from different methods. Don't worry, we'll be revisiting them very soon!

 The *Symbol*, *Security Type*, *Exchange* and *Currency* values are required for all instrument types. If your security type is *STK*, the values to the left are all you need. But if you're looking for the latest price on a Jan08 27.5 call, you need to give the method a bit more than that. I mean, it's really cool and can do a lot of things, but it can't read minds! The moral: be sure you include values in the appropriate fields based on what return values you want to get.

Once you have these parameters filled out to your satisfaction and click **OK**, you're basically sending a message to TWS asking to see market data for the specific contract. TWS will receive this message and reply with your requested market data. Without changing anything, let's use the data already in the sample app just to see what the response from TWS looks like. If you're ready, click **OK**.

TWS returns the market data values as shown in the screen below:



Note that the *id=1* corresponds directly to the Message ID of "1" in the Sample dialog.

EWrapper Methods that Return Market Data

These methods in the **EWrapper** interface return the following data:

tickPrice()

```
void tickPrice(int tickerId, int field, double price, int  
canAutoExecute)
```

tickSize()

```
void tickSize(int tickerId, int field, int size)
```

tickOptionComputation()

```
void tickOptionComputation(int tickerId, int field, double  
impliedVol, double delta, double modelPrice, double pvDividend)
```

tickGeneric()

```
void tickGeneric(int tickerId, int tickType, double value)
```

tickString()

```
void tickString(int tickerId, int tickType, String value)
```

tickEFP()

```
void tickEFP(int tickerId, int tickType, double basisPoints, String  
formattedBasisPoints, double impliedFuture, int holdDays, String  
futureExpiry, double dividendImpact, double dividendsToExpiry)
```

These methods are described in the *Java EWrapper Methods* section of the *API Reference Guide*.

Getting a Snapshot of Market Data

Another way to get market data from TWS to the Java Test Client is to get a snapshot of market data. A market data snapshot gives you all the market data in which you are interested for a contract for a single moment in time. What this means is that instead of watching the requested market data continuously scroll by in the *Market and Historical Data* text panel of the Java Test Client, you get a single "snapshot" of the data. This frees you from having to keep up with the scrolling data and having to cancel the market data request when you are finished.

To get snapshot market data, simply click the **Req Mkt Data** button, then fill in the appropriate fields in the Sample dialog, and finally check the *Snapshot* check box and click **OK**.

Snapshot is a parameter of the **reqMktData()** EClientSocket method.

Cancelling Market Data

Cancel Mkt Data

When you click the **Cancel Mkt Data** button, the attached ActionListener calls the **onCancelMktData()** method.

ActionListener for the Cancel Mkt Data button in createButtonPanel()

```
 JButton butCancelMktData = new JButton ( "Cancel Mkt Data") ;
butMktData.addActionListener ( new ActionListener() {
    public void actionPerformed ( ActionEvent e) {
        onCancelMktData();
    }
})
```

The **onCancelMktData()** method in turn calls the EClientSocket **cancelMktData()** method, and market data for the specified id is canceled:

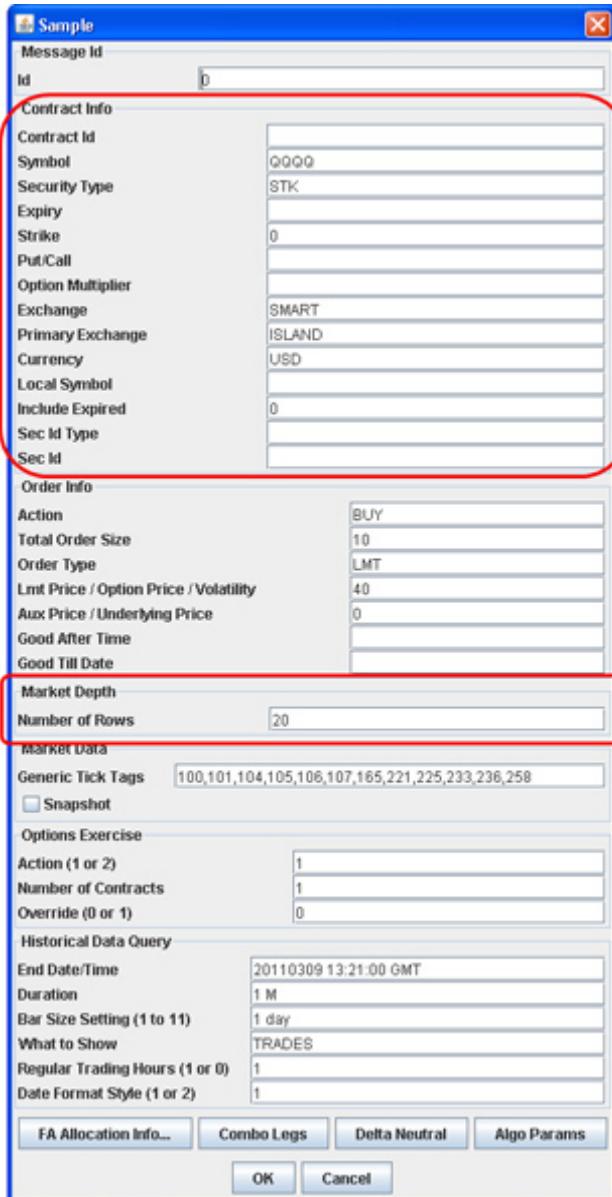
onCancelMktData() Method in SampleFrame.java

```
void onCancelMktData() {
// run m_orderDlg
m_orderDlg.show();
if( !m_orderDlg.m_rc ) {
    return;
// cancel market data
m_client.cancelMktData( m_orderDlg.m_id );
}
```

Chapter 8 - Requesting and Canceling Market Depth

This chapter discusses the methods for requesting and canceling market depth in the Java Test Client. We'll show you the methods and parameters behind the sample application and how they call the methods in the TWS Java API.

For requesting market depth, you need to use the highlighted fields in the Order Dialog as shown here:



What Happens When I Click the Req Mkt Depth Button?

Req Mkt Depth

When you click the **Req Mkt Depth** button, the attached ActionListener defined in **createButtonPanel()** in SampleFrame.java puts the appropriate method, **onReqMktDepth()**, into action.

ActionListener for the Req Mkt Depth button in createButtonPanel()

```
 JButton butMktData = new JButton ( "Req Mkt Data" );
butMktDepth.addActionListener ( new ActionListener() {
    public void actionPerformed ( ActionEvent e) {
        onReqMktDepth();
```

The method defined in the ActionListener, **onReqMktDepth()** is called, and the Order Dialog pictured below displays. Within that method, we make a call to the EClientSocket **reqMktDepth()** method below, which sends the values you entered in the shaded market data parameters to TWS.

onReqMktDepth() method in SampleFrame.java

```
void onReqMktDepth() {
    // run m_orderDlg
    m_orderDlg.show();
    if( !m_orderDlg.m_rc ) {
        return;
    }
    m_mktDepthDlg.setParams{ m_client, m_orderDlg.m_id)
    m_client.reqMktDepth( m_orderDlg.m_id, m_orderDlg.m_contract,
    xxxx m_orderDlg.m_marketDepthRows );
    m_mktDepthDlg.show();
}
```

The reqMktDepth() Method

Let's find out which parameters are used when you request market depth. The Class EClientSocket **reqMktDepth()** method header looks like this:

```
void reqMktDepth(int tickerId, Contract contract, int numRows)
```

Parameter	Description
tickerId	The ticker Id. Must be a unique value. When the market depth data returns, it will be identified by this tag. This is also used when canceling the market depth.
contract	This class contains attributes used to describe the contract.
numRows	Specifies the number of market depth rows to return.

This table is for illustrative purposes only and is not intended to portray valid API documentation.

As you can see from the previous table, this method has two parameters, *contract* and *numRows*, which correspond to the fields in the two sections of the Order dialog that you filled in.

The market depth will be returned via the **updateMktDepth()** and **updateMktDepthL2()** methods.

The **updateMktDepth()** and **updateMktDepthL2()** Methods

These EWrapper methods return market depth.

updateMktDepth() returns market depth.

```
void updateMktDepth(int tickerId, int position, int operation, int side, double price, int size)
```

updateMktDepthL2() returns Level II market depth.

```
void updateMktDepthL2(int tickerId, int position, String marketMaker, int operation, int side, double price, int size)
```

Cancelling Market Depth

Cancel Mkt Depth

When you click the **Cancel Mkt Depth** button, the attached ActionListener calls the **onCancelMktDepth()** method.

ActionListener for the Cancel Mkt Depth button in **createButtonPanel()**

```
JButton butCancelMktDepth = new JButton ( "Cancel Mkt Depth" );
butMktData.addActionListener ( new ActionListener() {
    public void actionPerformed ( ActionEvent e) {
        onCancelMktDepth();
```

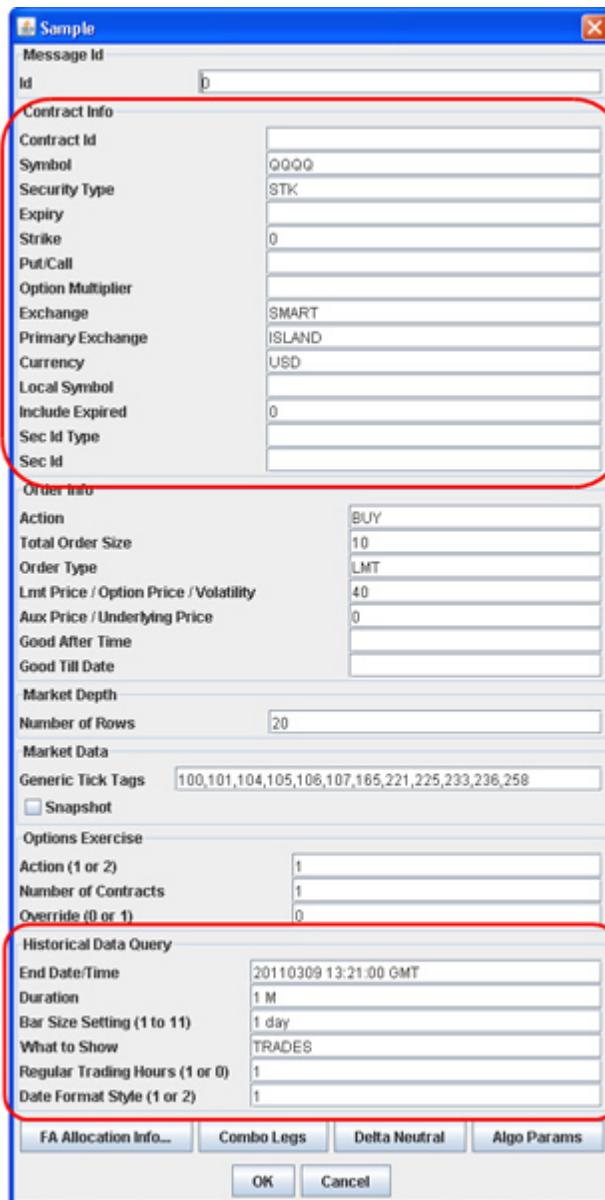
The **onCancelMktDepth()** method in turn calls the EClientSocket **cancelMktDepth()** method, and market depth for the specified id is canceled:

onCancelMktDepth() Method in **SampleFrame.java**

```
void onCancelMktDepth() {
// run m_orderDlg
m_orderDlg.show();
if( !m_orderDlg.m_rc ) {
    return;
// cancel market depth
m_client.cancelMktDepth( m_orderDlg.m_id );
}
```

Chapter 9 - Requesting and Canceling Historical Data

This chapter focuses on requesting and canceling historical data. We'll show you the methods and parameters behind the Java Test Client and how they call the methods in the TWS Java API. For requesting historical data, you need to use the fields circled below:



What Happens When I Click the Historical Data Button?

Historical Data

When you click the **Historical Data** button, the attached ActionListener defined in **createButtonPanel()** in SampleFrame.java calls the method **onHistoricalData()**.

ActionListener for the Historical Data button in createButtonPanel()

```
 JButton butHistoricalData = new JButton ( "Historical Data" );
butMktDepth.addActionListener ( new ActionListener() {
    public void actionPerformed ( ActionEvent e) {
        onHistoricalData();
```

The method defined in the ActionListener, **onHistoricalData()**, is called, and the familiar Order dialog appears. Within that method, we make a call to the EClientSocket **reqHistoricalData()** method below, which sends the values you entered to TWS.

onHistoricalData() method in SampleFrame.java

```
void onHistoricalData() {
// run m_orderDlg
m_orderDlg.show();
if( !m_orderDlg.m_rc ) {
    return;
}
// req historical data
m_client.reqHistoricalData( m_orderDlg.m_id,
m_orderDlg.m_contract, m_orderDlg.m_endDateTime,
m_orderDlg.m_durationStr, m_orderDlg.m_barSizeSetting,
m_orderDlg.m_whatToShow, m_orderDlg.m_useRTH,
m_orderDlg.m_formatDate );
```

The reqHistoricalData() Method

So which parameters are used when you request historical data? The parameters in the EClientSocket **reqHistoricalData()** method return the data you request. The **reqHistoricalData()** method header looks like this:

```
void reqHistoricalData (int id, Contract contract, String
endDateTime, String durationStr, String barSizeSetting, String
whatToShow, int useRTH, int formatDate)
```

This method has numerous parameters that correspond to the fields in the two sections of the Order dialog that you fill in, including end date and time, duration, bar size setting, what to show, regular trading hours, and date format style. There are too many to display the entire list of parameters and their values here, so you'll have to check out the *API Reference Guide* for more details.

The historicalData() Method

The values are returned via the parameters in the EWrapper interface **historicalData()** method, whose header is shown below.

```
void historicalData (int reqId, String date, double open, double
high, double low, double close, int volume, int count, double WAP,
boolean hasGaps)
```

You can see all of this method's parameters in the *historicalData()* method topic of the *API Reference Guide*.

Cancelling Historical Data

Cancel Hist. Data

When you click the **Cancel Hist. Data** button, the attached ActionListener calls the **onCancelHistoricalData()** method.

ActionListener for the Cancel Hist. Data button in createButtonPanel()

```
JButton butCancelHistoricalData = new JButton ( "Cancel Hist.
Data" );
butMktData.addActionListener ( new ActionListener() {
    public void actionPerformed ( ActionEvent e) {
        onCancelHistoricalData();
    }
});
```

The **onCancelHistoricalData()** method in turn calls the EClientSocket **cancelHistoricalData()** method, and historical data for the specified id is canceled:

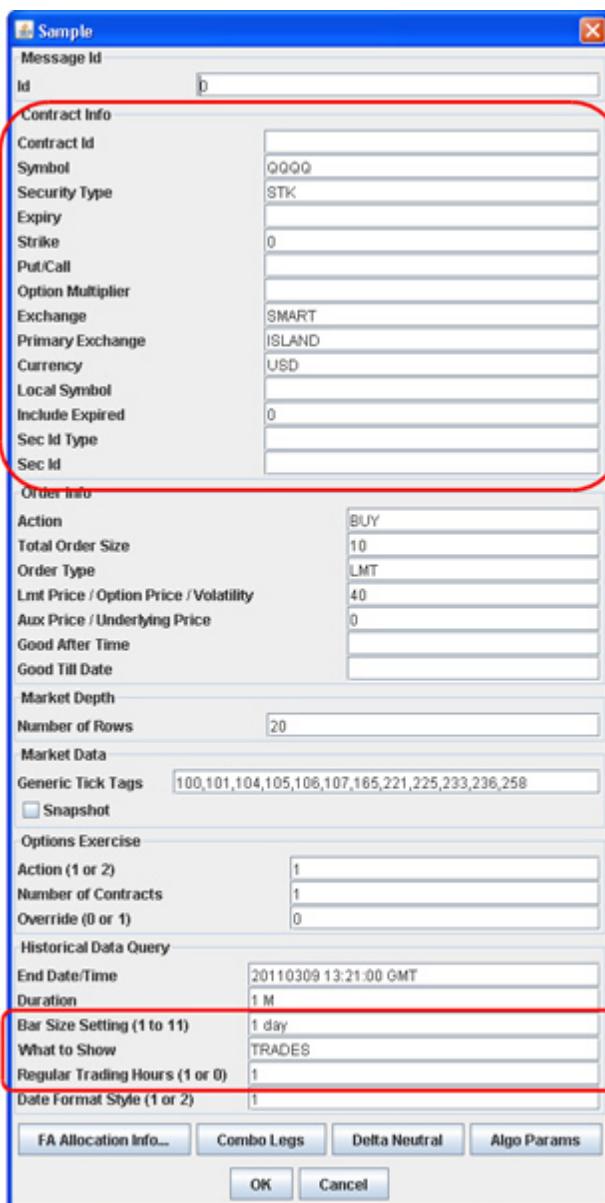
onCancelHistoricalData() Method in SampleFrame.java

```
void onCancelHistoricalData() {
    // run m_orderDlg
    m_orderDlg.show();
    if( !m_orderDlg.m_rc ) {
        return;
    // cancel historical data
    m_client.cancelHistoricalData( m_orderDlg.m_id );
    }
}
```

Historical data for the specified id is canceled.

Chapter 10 - Requesting and Canceling Real Time Bars

This chapter discusses the methods for requesting and canceling real time bars. Real time bars allow you to get a summary of real-time market data every five seconds, including the opening and closing price, and the high and the low within that five-second period (using TWS charting terminology, we call these five-second periods "bars"). You can also get data showing trades, midpoints, bids or asks. We show you the methods and parameters behind the Sample GUI, and how they call the methods in the TWS Java API. For requesting real time bars, you need to use the fields circled in the Order Dialog shown below:



What Happens When I Click the Req Real Time Bars Button?

Req Real Time Bars

When you click the **Req Real Time Bars** button, the attached ActionListener defined in **createButtonPanel()** in SampleFrame.java calls the method **onReqRealTimeBars()**.

ActionListener for the Req Real Time Bars button in createButtonPanel()

```
 JButton butRealTimeBars = new JButton( "Req Real Time Bars");
 butRealTimeBars.addActionListener( new ActionListener() {
     public void actionPerformed( ActionEvent e) {
         onReqRealTimeBars();
```

The method defined in the ActionListener, **onReqRealTimeBars()**, is called, and the familiar Order dialog appears. Within that method, we make a call to the EClientSocket **reqRealTimeBars()** method, which sends the values you entered to TWS (bar size setting, what to show, and whether or not to include data outside regular trading hours).

onReqRealTimeBars() method in SampleFrame.java

```
 void onReqRealTimeBars() {
    // run m_orderDlg
    m_orderDlg.show();
    if( !m_orderDlg.m_rc ) {
        return;
    }
    // req mkt data
    m_client.reqRealTimeBars( m_orderDlg.m_id,
xxxxm_orderDlg.m_contract,
      5 /* TODO: parse and use m_orderDlg.m_barSizeSetting */,
      m_orderDlg.m_whatToShow, m_orderDlg.m_useRTH > 0);
}
```

In the API release supported by this document, the real-time bars default to a size of five seconds. This means that no matter what you enter in the *Bar Size Setting* field in the Sample dialog, the size of the real-time bars you get will be five seconds.

The reqRealTimeBars() Method

The parameters in the EClientSocket **reqRealTimeBars()** method return the data you request. The **reqRealTimeBars()** method header looks like this:

```
 void reqRealTimeBars(int tickerId, Contract contract, int barSize,
 String whatToShow, boolean useRTH)
```

The realtimeBar() Method

The real time bars are returned via the parameters in the EWrapper interface **realtimeBar()** method, whose header is shown below.

```
void realtimeBar(int reqId, long time, double open, double high,
double low, double close, long volume, double wap, int count)
```

Cancelling Real Time Bars

Cancel Real Time Bars

To cancel real time bars, click the **Cancel Real Time Bars** button, then click **OK** in the Sample dialog. When you click **Cancel Real Time Bars**, the attached ActionListener calls the **onCancelRealTimeBars()** method.

ActionListener for the Cancel Real Time Bars button in createButtonPanel()

```
JButton butCancelRealTimeBars = new JButton ( "Cancel Real Time
Bars");
butMktData.addActionListener ( new ActionListener() {
    public void actionPerformed ( ActionEvent e) {
        onCancelRealTimeBars();
```

The **onCancelRealTimeBars ()** method in turn calls the EClientSocket **cancelRealTimeBars()** method, and data for real time bars for the specified id is canceled.

onCancelRealTimeBars() Method in SampleFrame.java

```
void onCancelRealTimeBars() {
// run m_orderDlg
m_orderDlg.show();
if( !m_orderDlg.m_rc ) {
    return;
// cancel market data
m_client.cancelRealTimeBars( m_orderDlg.m_id );
}
```

Chapter 11 - Subscribing to and Canceling Market Scanner Subscriptions

This chapter describes the methods used for requesting market scanner parameters, subscribing to a market scanner, and canceling a subscription to a market scanner. We'll show you the methods and parameters behind the Java Test Client sample application, and how they call the methods in the TWS Java API. In this case, the Scanner Dialog opens, instead of the Order Dialog which we've seen for the other buttons on the sample application.



What Happens When I Click the Market Scanner Button?

Market Scanner

When you click the **Market Scanner** button, the attached ActionListener defined in **createButtonPanel()** in SampleFrame.java calls the method **onScanner()**.

ActionListener for the Market Scanner button in createButtonPanel()

```
JButton butScanner = new JButton( "Market Scanner");
butScanner.addActionListener( new ActionListener() {
    public void actionPerformed( ActionEvent e) {
        onScanner();
```

The **onScanner()** method displays the Scanner Dialog (pictured on the previous page) and calls one of two EClientSocket methods, depending on which button you click.

onScanner() Method in SampleFrame.java

```
void onScanner() {
    m_scannerDlg.show();
    if (m_scannerDlg.m_userSelection ==
ScannerDlg.CANCEL_SELECTION) {
        m_client.cancelScannerSubscription(m_scannerDlg.m_id);
    }
    else if (m_scannerDlg.m_userSelection ==
ScannerDlg.SUBSCRIBE_SELECTION) {
        m_client.reqScannerSubscription(m_scannerDlg.m_id,
m_scannerDlg.m_subscription);
    }
    else if (m_scannerDlg.m_userSelection ==
ScannerDlg.REQUEST_PARAMETERS_SELECTION) {
        m_client.reqScannerParameters();
    }
}
```

If you click the **Request Parameters** button in the Scanner dialog, we make a call to the EClientSocket **reqScannerParameters()** method, which sends a request for available scanner parameters to TWS.

If you click the **Subscribe** button in the Scanner dialog, we make a call to the EClientSocket **reqScannerSubscription()** method, which sends the values you entered in the scanner parameters to TWS.

The reqScannerParameters() and reqScannerSubscription() Methods

reqScannerParameters() receives an XML document that describes the valid parameters that a scanner subscription can have. In the Java Test Client, these parameters are displayed in the TWS Server Responses text panel. The **reqScannerParameters()** method header looks like this:

```
public synchronized void reqScannerParameters()
```

reqScannerSubscription() receives market scanner results from TWS through the EWrapper method **scannerData()** method. The reqScannerSubscription method header looks like this:

```
public synchronized void reqScannerSubscription( int tickerId,  
ScannerSubscription subscription)
```

The scannerData() Method

The scanner data is returned from TWS by the EWrapper method **scannerData()**, whose header is shown below:

```
void scannerData(int reqId, int rank, ContractDetails  
contractDetails, String distance, String benchmark, String  
projection, String legsStr)
```

The scannerDataEnd() Method

There is one additional method in EWrapper used in conjunction with scanner subscriptions: **scannerDataEnd()**.

This method is called after a full snapshot of a scanner window has been received and serves as a sort of end tag. It helps define the end of one scanner snapshot and the beginning of the next.

Cancel a Market Scanner Subscription

To cancel your scanner subscription, click the **Cancel Subscription** button in the Scanner dialog. When you click this button, the attached ActionListener in ScannerDlg.java calls the **onCancelSubscription()** method.

ActionListener for the Cancel Subscription button

```
m_cancel.addActionListener( new ActionListener() {
    public void actionPerformed( ActionEvent e) {
        onCancelSubscription();
    }
})
```

The **onCancelSubscription()** method in turn calls the EClientSocket **cancelScannerSubscription()** method, and the scanner subscription for the specified id is canceled.

onCancelSubscription() Method in ScannerDlg.java

```
void onCancelSubscription() {
    m_userSelection = CANCEL_SELECTION;
    m_id = Integer.parseInt( m_Id.getText().trim() );
    setVisible( false );
}
```

The header for the EClientSocket method **cancelScannerSubscription()** is shown below.

```
void cancelScannerSubscription(int tickerId)
```

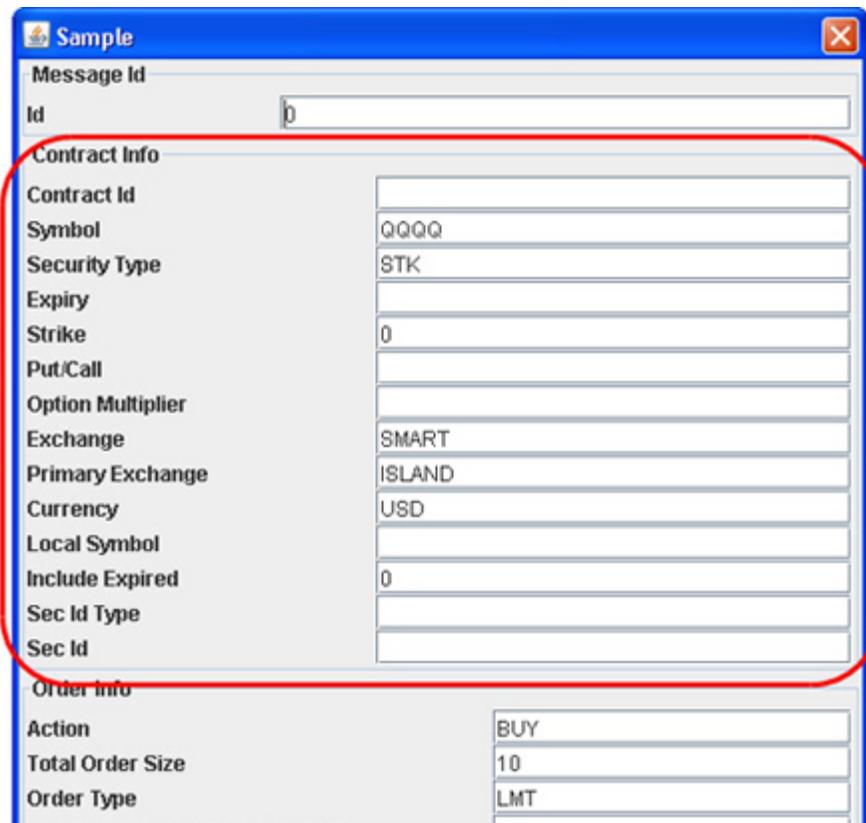
Chapter 12: Requesting Contract Data

This chapter shows you how to request contract data, including details such as the local symbol, conid, trading class, valid order types, and exchanges. We'll walk you through everything that happens from the time you click the **Req Contract Data** button in the sample application, to the moment you're taking in the fascinating details of your desired contract. It all happens fast, so pay attention!

What Happens When I Click the Req Contract Data Button?



To request contract data using the Java Test Client sample application, you'll need to enter data in the fields circled in the Order dialog pictured below. The Order dialog appears when you click the **Req Contract Data** button.



When you click the **Req Contract Data** button, the attached Action Listener in SampleFrame.java calls the **onReqContractData()** method, also in SampleFrame, which displays the Order dialog (the top half of which you can see in the picture above).

onReqContractData() Method in SampleFrame.java

```
void onReqContractData() {
    // run m_orderDlg
    m_orderDlg.show();
    if( !m_orderDlg.m_rc ) {
        return;
    }
    // req mkt data
    m_client.reqContractDetails( m_orderDlg.m_contract );
}
```

When you're finished entering the information in the Contract Info section of the Order dialog and you click **OK**, the **onReqContractData()** method calls the EClientSocket **reqContractDetails()** method.

The reqContractDetails() Method

The **reqContractDetails()** method, whose header is shown below, contains one parameter, *contract*. If you recall from earlier chapters, the *contract* parameter contains all the attributes used to describe the requested contract.

```
public synchronized void reqContractDetails(Contract contract)
```

The contractDetails() Method

The actual contract data is returned from TWS via the Java API EWrapper **contractDetails()** method. This method contains one parameter, *ContractDetails*, which you probably figured out by now contains all the attributes used to describe the requested contract.

```
void contractDetails(ContractDetails contractDetails)
```

In our Java Test Client sample application, the contract details you request are displayed in the TWS Server Responses text panel of the Sample GUI.

Orders and Executions

This section describes how the Java API sample application handles orders. We'll show you the methods, events and parameters behind such trading tasks as placing and canceling orders, exercising options and viewing open orders and executions.

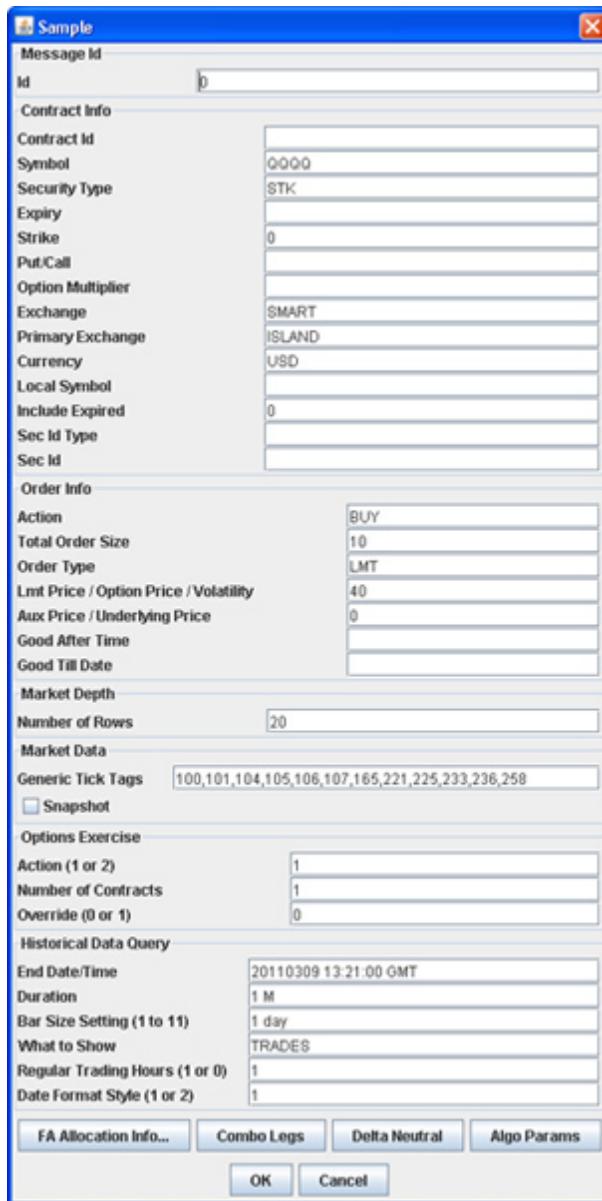
Here's what you'll find in this section:

- [Chapter 13: Placing and Canceling an Order](#)
- [Chapter 14: Exercising Options](#)
- [Chapter 15: Extended Order Attributes](#)
- [Chapter 16: Requesting Open Orders](#)
- [Chapter 17 Requesting Executions](#)

Using the Java Test Client is a good way to practice locating and using the reference information in the API Reference Guide. With the sample program, you can compare the data in the sample message with the method parameters in the API Reference Guide.

Chapter 13: Placing and Canceling an Order

These next few chapters look at the order-related actions in the Java Test Client sample application, which are grouped together in the second series of buttons. Actions such as placing and Canceling an order, exercising options and applying extended order attributes are included here. Just as in earlier chapters where we discussed some of the other buttons in the sample application, when you click one of these order-related buttons, the Sample dialog appears.



What Happens When I Place an Order?

Let's take a look at what happens when you place an order. In this section, we'll show you the methods and parameters behind the Java Test Client sample application, and how they call the methods in the TWS Java API.



When you click the **Place Order** button, the attached ActionListener defined in **createButtonPanel()** in SampleFrame.java calls the method **onPlaceOrder()**, also in SampleFrame, and displays the Order dialog.



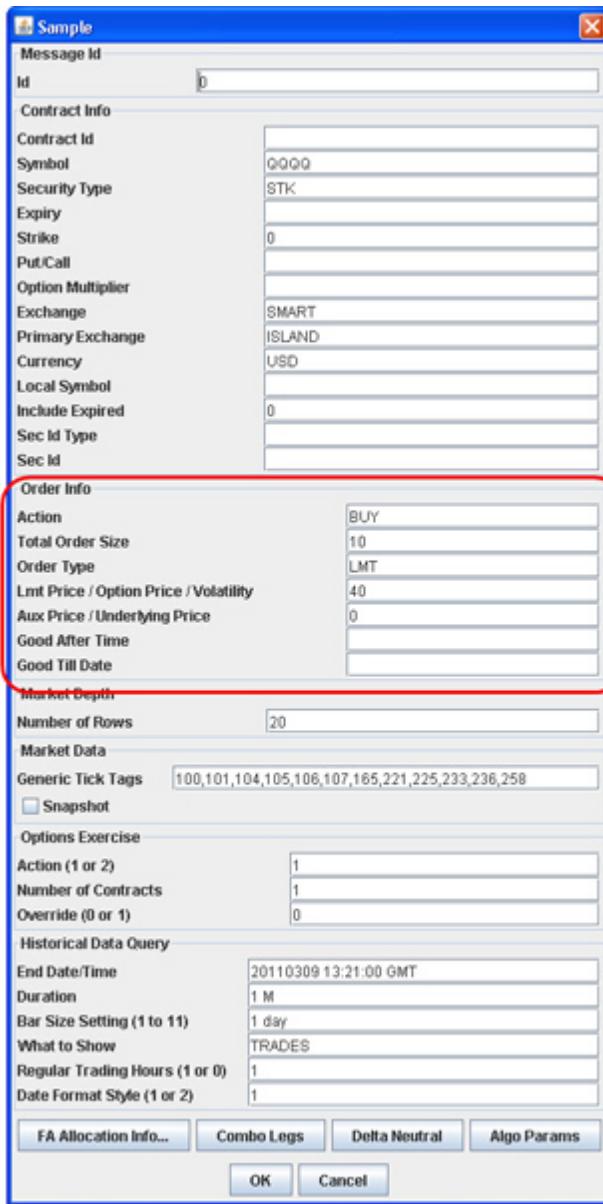
If you've read some of the previous chapters, you'll know that each button in the sample application is associated with an Action Listener in SampleFrame.java. Each Action Listener in turns calls another method in SampleFrame, which in turn calls a method in the Java API EClientSocket class. For the rest of the buttons in the sample application, we'll skip showing you each and every Action Listener. However, you can always take a peek at [Appendix E](#), which lists the buttons in the sample application and their Action Listeners.

As we mentioned earlier, the Order dialog is actually the orderDlg object, an extension of Jdialog, which is instantiated when the method in the ActionListener executes. This dialog makes it easy to define the valid values for the parameters that will be sent to TWS via the EClientSocket methods.

When placing an order, you'll use the order fields circled in the Order dialog pictured on the next page.

Orders and Executions

Chapter 13: Placing and Canceling an Order



The **onPlaceOrder()** method in turn calls yet another method in SampleFrame called **placeOrder()**, which displays the Order dialog and calls the Java API EClientSocket method **placeOrder()**.

Be careful not to confuse the two **placeOrder()** methods; one is part of the Java Test Client (the one that displays the Order dialog) and the other is part of the Java API (the one in EClientSocket that sends your order information to TWS).

placeOrder() Method in SampleFrame.java

```

void placeOrder(boolean whatIf) {
    This is the part of the method that displays the Order dialog:
    // run m_orderDlg
    m_orderDlg.show();
    if( !m_orderDlg.m_rc ) {
        return;
    }
    Order order = m_orderDlg.m_order;
    // save old and set new value of whatIf attribute
    boolean savedWhatIf = order.m_whatIf;
    order.m_whatIf = whatIf;

    This is the part of the method that calls the EClientSocket
    placeOrder() method:
    // place order
    m_client.placeOrder( m_orderDlg.m_id, m_orderDlg.m_contract,
    order );
    // restore whatIf attribute
    order.m_whatIf = savedWhatIf;
}

```

The placeOrder() Method

The EClientSocket **placeOrder()** method, shown below, sends the values you entered in the Order dialog to TWS.

```

public synchronized void placeOrder( int id, Contract contract,
Order order)

```

Parameter	Description
id	The order Id. You must specify a unique value. When the order status returns, it will be identified by this tag. This tag is also used when canceling the order.
contract	This class contains attributes used to describe the contract.
order	This structure contains the details of the order. Note: Each client MUST connect with a unique clientId.

Tables are for illustrative purposes only and are not intended to represent valid API information.

The *contract* and *order* classes contain the parameters that correspond to the *contract* and *order* fields in the Order dialog that you fill in.

The **orderStatus()** Method

The values are returned via the EWrapper **orderStatus()** method, whose header is shown below.

```
void orderStatus(int orderId, String status, int filled, int
remaining, double avgFillPrice, int permId, int parentId, double
lastFillPrice, int clientId, String whyHeld)
```

Cancelling an Order

Cancel Order

To cancel an order, click the **Cancel Order** button in the Sample dialog. When you click this button, the attached ActionListener in SampleFrame.java calls the **onCancelOrder()** method.

The **onCancelOrder()** method displays the Order dialog, in which you must enter the correct *Id*, then press **OK** to completely cancel your order. Behind the scenes, the **onCancelOrder()** method calls the EClientSocket **cancelOrder()** method, and the order associated with the specified ID is canceled.

onCancelOrder() Method in ScannerDlg.java

```
void onCancelOrder() {
    // run m_orderDlg
    m_orderDlg.show();
    if( !m_orderDlg.m_rc ) {
        return;
    }
    // cancel order
    m_client.cancelOrder( m_orderDlg.m_id );
}
```

The header for the EClientSocket method **cancelOrder()** is shown below.

```
void cancelOrder(int id)
```

Modifying an Order

To modify an order using the API, resubmit the order you want to modify using the same order id, but with the price or quantity modified as required. Only certain fields such as price or quantity can be altered using this method. If you want to change the order type or action, you will have to cancel the order and submit a new order.

Requesting "What-If" Data before You Place an Order

What If

Another feature supported by the Java Test Client sample application is the ability to request margin and commission "what if" data before you place an order. This means that you can click the **What If** button, set up your order as if you were actually placing it, then see what the margins and commissions would be if the trade went through.

The *Order* class, as you recall from earlier in this chapter, is one of the parameters in the **placeOrder()** EClientSocket method. Within the order class, there is an attribute called *whatIf()*. When this value is set to true, the margin and commission data is received via the *OrderState* class, which is one of the parameters in the **openOrder()** EW wrapper method.

Chapter 14: Exercising Options

This chapter discusses how the Java Test Client sample application exercises options prior to expiration, and instructs options to lapse. We'll show you the methods and parameters behind the Options Exercise area of the sample application, and see how these methods call the methods in the TWS Java API.

What Happens When I Click the Exercise Options Button?

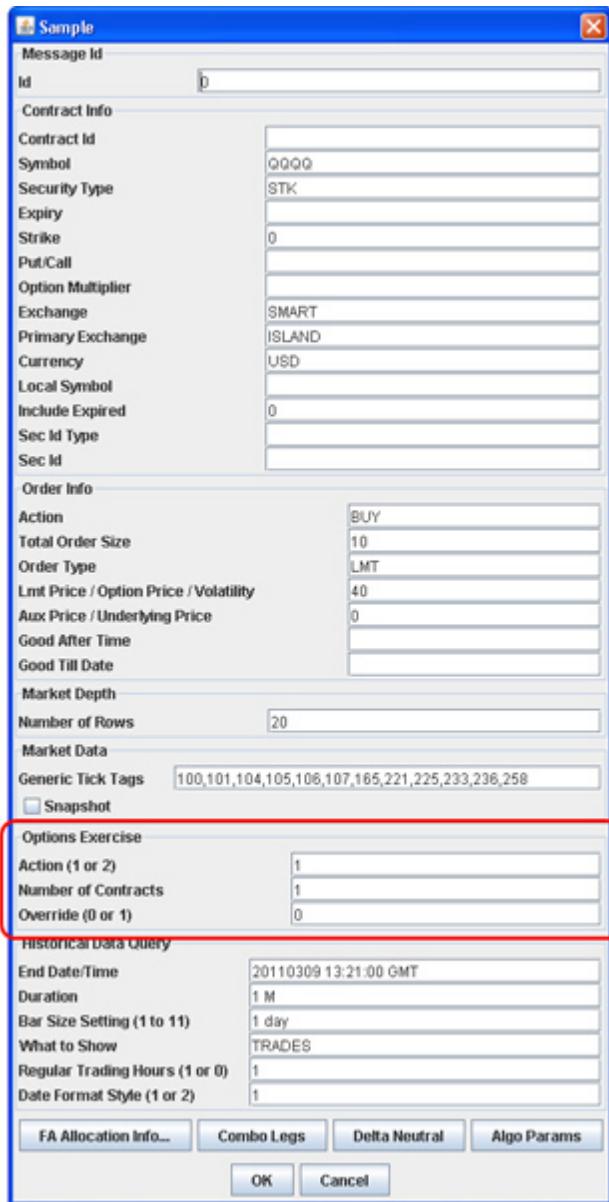
Exercise Options

When you click the **Exercise Options** button, the attached ActionListener defined in **createButtonPanel()** in SampleFrame.java calls the method **onExerciseOptions()**, also in SampleFrame, which displays the Order dialog.

onExerciseOptions() Method in SampleFrame.java

```
void onExerciseOptions() {
    m_orderDlg.show();
    if( !m_orderDlg.m_rc ) {
        return;
    }
    // cancel order
    m_client.exerciseOptions( m_orderDlg.m_id,
    m_orderDlg.m_contract, m_orderDlg.m_exerciseAction,
    m_orderDlg.m_exerciseQuantity, m_orderDlg.m_order.m_account,
    m_orderDlg.m_override);
}
```

To exercise an option, you'll use the fields circled in the Order dialog, pictured on the next page.



Within the **onExerciseOptions()** method, we also make a call to API EClientSocket **exerciseOptions()** method, shown below, which sends the values you entered in Order dialog to TWS.

The exerciseOptions() Method

```
public synchronized void exerciseOptions( int tickerId, Contract  
contract, int exerciseAction, int exerciseQuantity, String account,  
int override)
```

Parameter	Description
tickerId	The Id for the exercise request
contract	This class contains attributes used to describe the contract.
exerciseAction	this can have two values: <ul style="list-style-type: none">• 1 = exercise• 2 = lapse
exerciseQuantity	The number of contracts to be exercised
account	For institutional orders. Specifies the IB account.
override	Specifies whether your setting will override the system's natural action. For example, if your action is "exercise" and the option is not in-the-money, by natural action the option would not exercise. If you have override set to "yes" the natural action would be overridden and the out-of-the money option would be exercised. Values are: <ul style="list-style-type: none">• 0 = do not override• 1 = override

Tables are for illustrative purposes only and are not intended to represent valid API information.

In this case, no values are returned by the EWrapper interface, as is the case with many other functions in the TWS Java API.

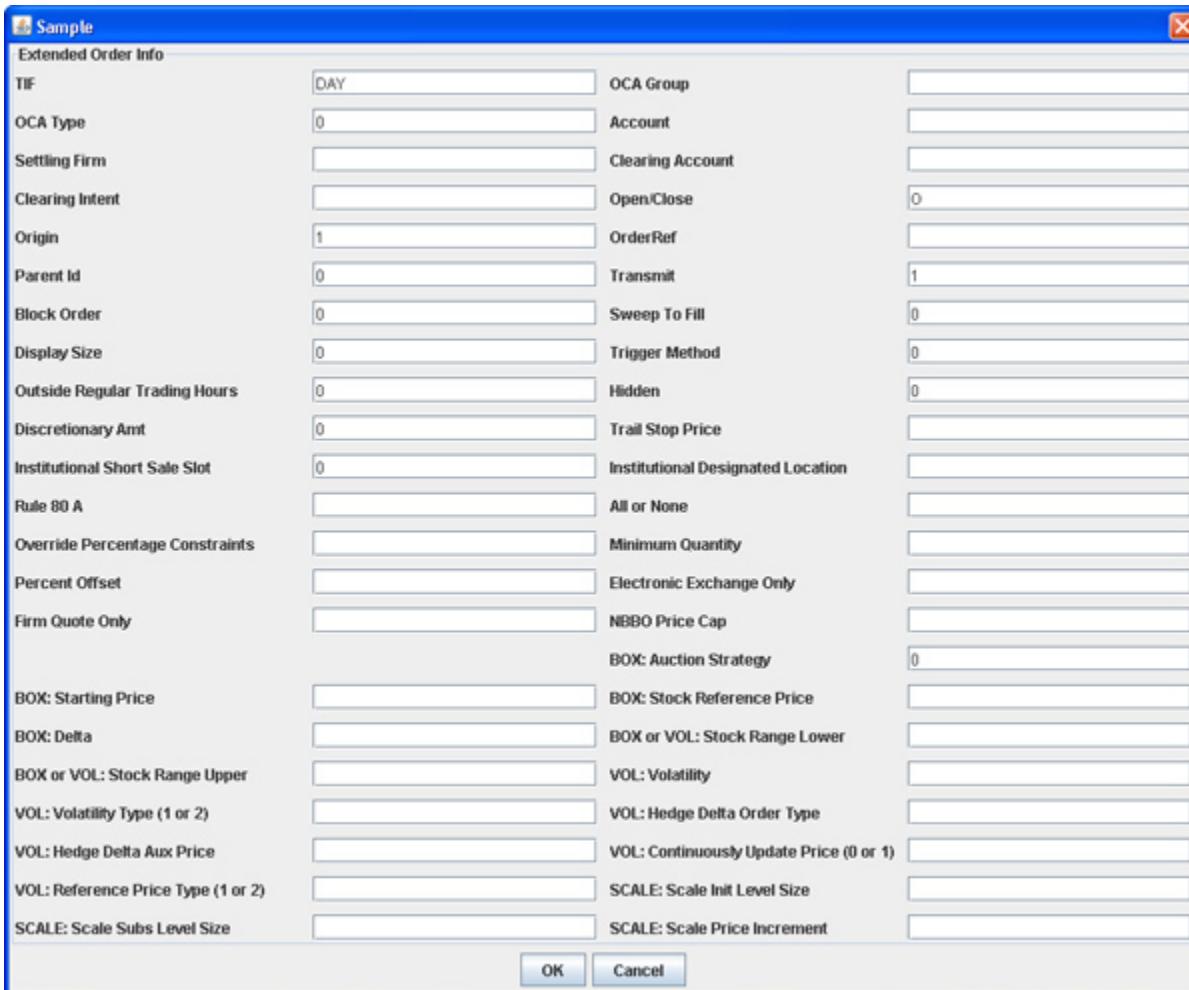
Chapter 15: Extended Order Attributes

This chapter discusses how to apply extended, or non-essential, order attributes to your order. This sample action is different from many of the others we've looked at, as the extended order attributes for the Java API are actually included in the *Order* java class. For ease of use, we have created a separate dialog in which you can assign values to the extended order attributes. So although you will see a new dialog when you click the **Extended** button, the selections you're setting do not come from a new API method.

What Happens When I Click the Extended Button?

Extended

Let's take a look at how this functionality is set up by taking a look at what happens when we click the **Extended** button. The attached Action Listener in SampleFrame.java calls the **onExtendedOrder()** method, also in SampleFrame, and **onExtendedOrder()** in turn displays the Extended Order dialog, shown below.



The fields in this dialog are actually attributes of the *Order* class, which is called in the **placeOrder()** method.

onExtendedOrder() Method in SampleFrame.java

```
void onExtendedOrder() {
    //Show the extended order attributes dialog
    m_extOrdDlg.show();
    if( !m_extOrdDlg.m_rc ) {
        return;
    }
    // Copy over the extended order details
    copyExtendedOrderDetails( m_orderDlg.m_order,
    m_extOrdDlg.m_order);
}
```

So this time, within the **onExtendedOrder()** method, we call the private **copyExtendedOrderDetails()** method, also in SampleFrame.

copyExtendedOrderDetails() copies the values you enter in this dialog to the parameters to the EClientSocket *Order* class when you click **OK** in the Extended Order dialog.

copyExtendedOrderDetails() Method in SampleFrame.java

```
private void copyExtendedOrderDetails( Order destOrder, Order
srcOrder)
```

That's all the **Extended** button does. Until you place an order, the extended attributes are just that - static, sitting, lazy, waiting attributes. But once you create and place an order, the values you entered/modified in the Extended Order dialog are used in your order, and will continue to be applied to every order until you change them.

Chapter 16: Requesting Open Orders

In this chapter, we're going to take a look at three related methods/buttons in the sample application:

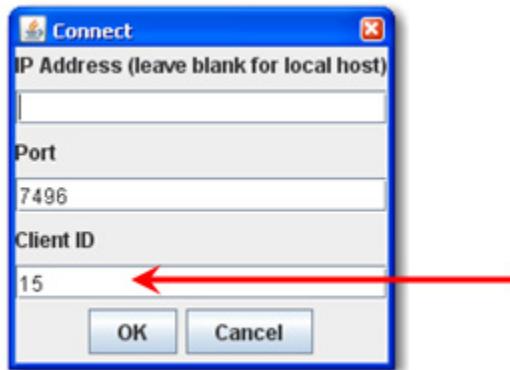
- **Request Open Orders**
- **Request All Open Orders**
- **Request Auto Open Orders**

How are they related?

Well, obviously they all give you information about open orders. The difference between them is the Client ID, which you set (or not!) when you connect to TWS.

Running Multiple API Sessions

You can connect up to eight API sessions to one TWS client, but the catch is that you have to assign a new client ID for each API session. Therefore, any orders sent from these clients can be tracked through the life of the order, and everyone knows where they came from and who's responsible for them. So be careful!:



If you happen to have TWS up and running now and want to try this out, simply run multiple sample API sessions as described in the following steps:

- 1 Click the **Connect** button and connect to the first session. Note that the *Client ID* is set to "0."
- 2 Do the same for another session. If you don't change anything, you'll see that you are not able to connect to this second session. In the *Errors and Messages* text panel on the sample application, the API will kindly tell you "Already connected."
- 3 Now try it with a unique Client ID. Click **Connect** again, only this time type **1** (or any other unique Client ID) in the *Client ID* field, then click **OK**.

The Difference between the Three Request Open Orders Buttons

Now you're ready to learn the difference between the three Request Open Orders methods/buttons:

- **Request Open Orders** shows you any open orders made from that client, and if it's the "0" client ID client, you'll also see open orders sent from TWS.
- **Request All Open Orders** method shows you open orders sent from ALL clients connected to TWS, and all open orders that were sent from that TWS.
- **Request Auto Open Orders** method can only be used by the API with the client ID of "0." Clicking this button sets the boolean parameter to "True" and forever binds TWS orders to the API client. From that day forward, any time an open order exists on TWS it will automatically be returned via the EWrapper methods, and in this case be displayed in the TWS Server Responses text panel of the sample application.

Got all that? Good, let's see the details.

What Happens When I Click the Req Open Orders Button?

Req Open Orders

When you click the **Req Open Orders** button, the attached Action Listener in SampleFrame.java calls the **onReqOpenOrders()** method, also in SampleFrame, and shown below.

onReqOpenOrders() Method in SampleFrame.java

```
void onReqOpenOrders() {  
    m_client.reqOpenOrders();  
}
```

The **onReqOpenOrders()** method calls the API EClientSocket **reqOpenOrders()** method.

The reqOpenOrders() Method

The **reqOpenOrders()** method gets all open orders that were sent from your API client, and if you have a Client ID of "0" it gets the TWS orders as well.

```
public synchronized void reqOpenOrders()
```

The open order information is returned via the **openOrder()** and **orderStatus()** methods, which are defined in the EWrapper interface.

In the Java Test Client sample application, you will see the open order info displayed in the *TWS Server Responses* text panel.

What Happens When I Click the Req All Open Orders Button?

Req All Open Orders

When you click the **Req All Open Orders** button, the attached Action Listener in SampleFrame.java calls the **onReqAllOpenOrders()** method, also in SampleFrame, as shown below.

onReqAllOpenOrders() Method in SampleFrame.java

```
void onReqAllOpenOrders() {  
    // request list of all open orders  
    m_client.reqAllOpenOrders();  
}
```

The **onReqAllOpenOrders()** method calls the API EClientSocket **reqAllOpenOrders()** method.

The reqAllOpenOrders() Method

The **reqAllOpenOrders()** method gets all open orders that were sent from your API client, and if you have a Client ID of "0" it gets the TWS orders as well.

```
public synchronized void reqAllOpenOrders()
```

If you only have a single API client running, nothing will happen when you click this button. It's only useful if you are running multiple API clients off the same TWS session. In that case, this method gets all open orders from all clients and TWS, and returns them via the **openOrder()** and **orderStatus()** methods, which are defined in the EWrapper interface.

In the Java Test Client sample application, you will see the open order information displayed in the *TWS Server Responses* text panel.

What Happens When I Click the Req Auto Open Orders Button?

Req Auto Open Orders

Last but not least is the **Req Auto Open Orders** button. When you click this button, , the attached Action Listener in SampleFrame.java calls the **onReqAllOpenOrders()** method, also in SampleFrame, as shown below.

onReqAutoOpenOrders() Method in SampleFrame.java

```
void onReqAutoOpenOrders() {  
    // request to automatically bind any newly entered TWS orders  
    // to this API client. NOTE: TWS orders can only be bound to  
    // client's with clientId=0.  
    m_client.reqAutoOpenOrders( true );  
}
```

The **onReqAllOpenOrders()** method calls the API EClientSocket **reqAutoOpenOrders()** method.

The reqAutoOpenOrders() Method

```
public synchronized void reqAutoOpenOrders()
```

This method has a single parameter:

- *bAutoBind*: If set to TRUE, newly created TWS orders will be associated with the client. If set to FALSE, no association will be made.

As we mentioned above, the **onReqAutoOpenOrders()** method calls the **reqAutoOpenOrders()** method in EClientSocket and sets the *bAutoBind* parameter in that method to true. That is, if you are using an API with a Client ID of "0." Otherwise you'll receive an error message and the auto binding won't be enabled.

But if your Client ID is "0", you have just bound all future TWS open orders to your client. This means that any time an order is sent from that TWS it will automatically be fed back through the EWrapper methods and show up like magic in the TWS Server Responses text panel.

In the Java Test Client sample application, you will see the open order information displayed in the *TWS Server Responses* text panel.

Chapter 17 Requesting Executions

This chapter shows you how to request execution reports using the Filter Criteria dialog in the Java Test Client sample application. You can retrieve all execution reports, or only those you want by entering specific criteria such as time, symbol, exchange and more. We'll show you the methods and parameters behind the sample application, and how they call the methods in the TWS Java API.

What Happens When I Click the Req Executions Button?

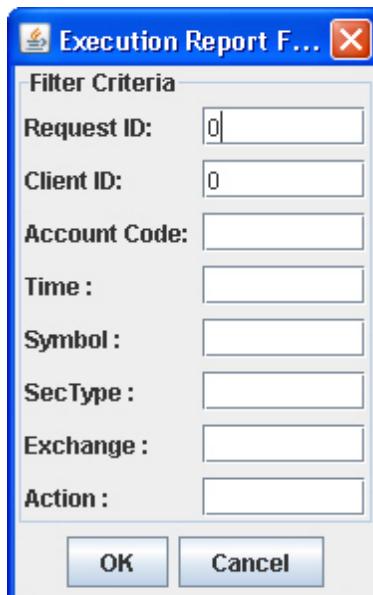
Req Executions

When you click the **Req Executions** button, the attached Action Listener in SampleFrame.java calls the **onReqExecutions()** method, also in SampleFrame.

onReqExecutions() Method in SampleFrame.java

```
void onReqExecutions() {  
    ExecFilterDlg dlg = new ExecFilterDlg(this);  
    dlg.show();  
    if ( dlg.m_rc ) {  
        // request execution reports based on the supplied filter  
        criteria  
        m_client.reqExecutions( dlg.m_execFilter );  
    }  
}
```

The **onReqExecution()** method displays the Execution Filter dialog shown below:



Note that the *Client ID* field comes with a default value of "0." This isn't by chance! You can leave all of the other fields blank and everything will be fine. But if you leave the *Client ID* field blank, you'll get nothing, no matter what other field values you may enter. After you define the filter criteria and click **OK**, we make a call to the **reqExecutions()** method in the Java API's EClientSocket.

The reqExecutions() Method

The **reqExecutions()** method in EClientSocket sends the values you entered in the Execution Filter dialog to TWS. Another way of saying this is that the filter criteria you entered in the Execution Filter dialog are the parameters for this method.

```
public synchronized void reqExecutions(ExecutionFilter filter)
```

The execDetails() Method

The execution data are returned via the **execDetails()** method in the Java API EWrapper interface.

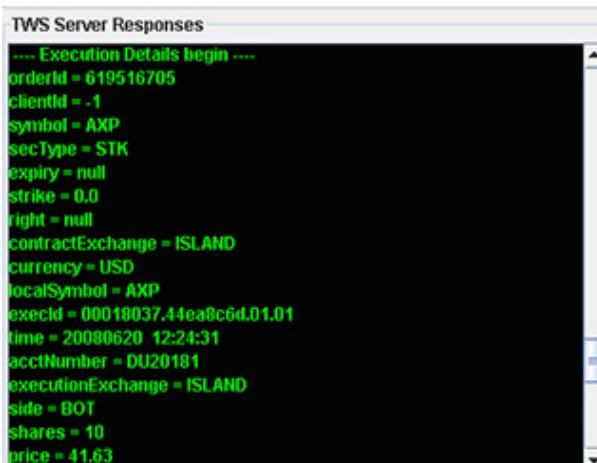
```
void execDetails( int orderId, Contract contract, Execution
execution)
```

As you can see from the method declaration above, the **execDetails()** method contains the following parameters:

Parameter	Description
orderId	The order Id that was specified previously in the call to placeOrder().
contract	This structure contains a full description of the contract that was executed.
execution	This structure contains addition order execution details.

Tables are for illustrative purposes only and are not intended to represent valid API information.

In the Java Test Client sample application, you will see the execution information you requested displayed in the *TWS Server Responses* text panel:



```
.... Execution Details begin ....
orderId = 619516705
clientId = -1
symbol = AXP
secType = STK
expiry = null
strike = 0.0
right = null
contractExchange = ISLAND
currency = USD
localSymbol = AXP
execId = 00018037.44ea8c6d.01.01
time = 20080620 12:24:31
acctNumber = DU20181
executionExchange = ISLAND
side = BOT
shares = 10
price = 41.63
```

Additional Tasks

This section describes some additional tasks that you can perform using the Java API sample application. We'll show you the methods, events and parameters behind such tasks as requesting the current server time, the next ID, subscribing and unsubscribing to news bulletins, and changing the server logging level.

Here's what you'll find in this section:

- [Chapter 18 - Requesting the Current Time](#)
- [Chapter 19: Subscribing to News Bulletins](#)
- [Chapter 20: View and Change the Server Logging Level](#)



In addition to the tasks described in this chapter, the Java API sample application also includes a few more advanced functions, including the ability to calculate volatility and option price, and support for IBAlgos. For more information on these and other advanced capabilities of the Java API, see our API Reference Guide, available from the Reference Guide tab on our IB API web page.

Chapter 18 - Requesting the Current Time

This chapter discusses the method for requesting the current system time. Actually, "discusses" is really not the correct word. It merely "states" the method, which is quite solitary with no parameters to call its own.

What Happens When I Click the Req Current Time Button?

Req Current Time

When you click the **Req Current Time** button, the attached ActionListener calls the method `onScanner()`.

ActionListener for the Req Current Time button in `createButtonPanel()`

```
 JButton butCurrentTime = new JButton( "Req Current Time") ;
butCurrentTime.addActionListener( new ActionListener() {
    public void actionPerformed( ActionEvent e) {
        onReqcurrentTime();
    }
}
```

`onReqcurrentTime()` method, shown below, is called, and within that method, without ever displaying that Order Dialog, we make a call to API EClientSocket `reqcurrentTime()` method, which sends a request to TWS for the current server time.

`onReqcurrentTime()` Method

```
 void onReqcurrentTime() {
    m_client.reqcurrentTime();
}
```

Some methods have no parameters, but they're included anyway. The EClientSocket `reqcurrentTime()` method is one of these methods. It has no parameters; it simply request the current server time. The `reqcurrentTime()` method looks like this:

```
 void reqcurrentTime()
```

The time is returned by the EWrapper method `currentTime()`, which as you might have already suspected, contains only one parameter: `time`. The `currentTime()` method looks like this:

```
 void currentTime(long time)
```

Time's up! Actually, we should probably request the current time before we jump to that conclusion. But it IS time to move on to the order-related methods, regardless of what the EWrapper has to say!

Chapter 19: Subscribing to News Bulletins

This chapter shows you how to subscribe to IB news bulletins through the Java Test Client. Once you subscribe, all bulletins will display in the *TWS Server Responses* text panel of the sample application. The news bulletins keep you informed of important exchange disruptions.

We will show you the methods and parameters behind the news bulletin feature of the Java Test Client sample application, and how they call the methods in the TWS Java API.

What Happens When I Click the Req News Bulletins Button?

Req News Bulletins

When you click the **Req News Bulletin** button, the attached Action Listener in SampleFrame.java calls the **onReqNewsBulletins()** method, also in SampleFrame.

onReqNewsBulletins() Method in SampleFrame.java

```
void onReqNewsBulletins() {
    // run m_newsBulletinDlg
    m_newsBulletinDlg.show();
    if( !m_newsBulletinDlg.m_rc ) {
        return;
    }
    if ( m_newsBulletinDlg.m_subscribe ) {
        m_client.reqNewsBulletins( m_newsBulletinDlg.m_allMsgs);
    }
    else {
        m_client.cancelNewsBulletins();
    }
}
```

The **onReqNewsBulletins()** method displays the News Bulletin Subscription dialog shown below.



When you click **Subscribe** in this dialog, we call the Java API EclientSocket **reqNewsBulletins()** method.

The reqNewsBulletins() method

This EClientSocket method tells TWS that you want to subscribe to news bulletins.

```
public synchronized void reqNewsBulletins( boolean allMsgs)
```

reqNewsBulletins() has one parameter: *allMsgs*. If you click the *receive new messages only* radio button in the News Bulletin Subscription dialog, the *allMsgs* parameter, which asks "receive ALL messages, old and new?" will be set to false, which basically means "no, thanks!". If you select *receive all the current day's messages and any new messages*, the *allMsgs* parameter gets set to true, which means, "yes, please!" Pretty cool, right? Either way, you are now subscribed to news bulletins, and either way you will receive any NEW bulletins that get posted from the time you subscribe.

The updateNewsBulletin() Method

The bulletins are returned via the **updateNewsBulletin()** method in the Java API EWrapper interface. The header for this method is shown below.

```
void updateNewsBulletin(int msgId, int msgType, String message,
String origExchange)
```

updateNewsBulletin() contains the following parameters:

Parameter	Description
msgId	The bulletin ID, incrementing for each new bulletin.
msgType	Specifies the type of bulletin. Valid values include: <ul style="list-style-type: none">• 1 = Regular news bulletin• 2 = Exchange no longer available for trading• 3 = Exchange is available for trading
message	The bulletin's message text.
origExchange	The exchange from which this message originated.

Tables are for illustrative purposes only and are not intended to represent valid API information.

Cancelling News Bulletins

If you're tired of knowing what's going on around you, you can elect to unsubscribe, or cancel the news bulletins. To unsubscribe to news bulletin, you first need to click the **Req News Bulletins** button in the Java Test Client sample application. Then you click **Unsubscribe** in the News Bulletin Subscription dialog, and we call the EClientSocket **cancelNewsBulletins()** method, which as the name implies, cancels your news bulletin subscription.

The **cancelNewsBulletins()** method header looks like this:

```
public synchronized void cancelNewsBulletins()
```

Because you are simply canceling a request, there are no values returned by this method.

Chapter 20: View and Change the Server Logging Level

This chapter shows you how to view and change the server logging level.

As client requests are processed (both system and API clients), TWS logs certain information to its log.txt log file located in the installation directory. The purpose of this file is to help resolve problems by providing some insight into the state of the program before the problem occurred. In the Java Test Client sample application, you can specify how detailed the information will be when entered into the log.txt file. Basically, the higher the log level, the more performance overhead that may be incurred. By default, the server logging level is set to "2" for error logging.



*See our API Reference Guide for more information about API logging. The API Reference Guide is available from the **Application Programming Interfaces** page on our web site as an online guide or a downloadable/printable PDF.*

What Happens When I Click the Server Logging Button?

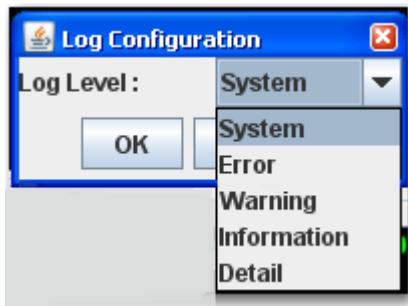
Server Logging

Anyway, to see or change the server logging level, you first click the **Server Logging** button on the Java Test Client. As with all the other buttons on the sample application, when you click this button, the attached Action Listener in SampleFrame.java calls the **onServerLogging()** method, also in SampleFrame.

onServerLogging() Method in SampleFrame.java

```
void onServerLogging() {
    // get server logging level
    LogConfigDlg dlg = new LogConfigDlg( this );
    dlg.show();
    if( !dlg.m_rc) {
        return;
    }
    // connect to TWS
    m_client.setServerLogLevel( dlg.m_serverLogLevel );
}
```

The **onServerLogging()** method displays the Log Configuration dialog shown on the next page.



The default level appears in the *Log Level* field. We've expanded the dropdown list in the figure above just to show you what log levels you can choose. Once you select a level and click **OK**, we call the EClientSocket **setServerLogLevel()** method.

The **setServerLogLevel()** Method

The EClientSocket method **setServerLogLevel()** contains a single parameter, *logLevel*.

```
public synchronized void setServerLogLevel(int LogLevel)
```

The *logLevel* parameter specifies the level of log entry detail used by TWS when processing API requests. The valid values for this parameter correspond to your choices in the Log Level dropdown in the Log Configuration dialog:

- 1 = SYSTEM
- 2 = ERROR
- 3 = WARNING
- 4 = INFORMATION
- 5 = DETAIL

This concludes our discussion of the Java Test Client sample application for individual accounts. The next chapter discusses methods used for Financial Advisor and multi-client accounts.

Additional Tasks

Chapter 20: View and Change the Server Logging Level

Sample Applications for the Java API

We've included two samples to help you get a better feel for what you can do with the Java API. For each sample, we've included a description of its primary purpose and a walkthrough of the actual Java code in the sample. The first sample is very simple, and the second sample is somewhat more complicated.

The sample code for the examples in this section can be found on our web site at http://www.interactivebrokers.com/en/p.php?f=programInterface&ib_entity=llc.

The following chapters are included in this section:

- [Chapter 21 - Downloading and Preparing the Sample Code](#)
- [Chapter 22 - Example 1: Requesting Market Data](#) - In this simple sample program, we show you how to get the last price for a given symbol.
- [Chapter 23 - Example 2: Automating Option Orders](#) - In this sample program, we look at volatility for a given underlying to determine whether or not to place a straddle order.



In this section, we do not go into detail about Java-specific programming instructions. In other words, we assume that you already have experience programming in Java.

Chapter 21 - Downloading and Preparing the Sample Code

This chapter describes how to get the Java code for the samples in this section, and how to open the samples in NetBeans as a new project.



While you are always free to use your own favorite Java IDE, we've written these sample chapters with NetBeans in mind. Where necessary, we've included minimal instruction on how to set up and run the samples in NetBeans. Be sure to adapt those instructions to the Java IDE that you are using.

Download the Samples

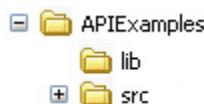
Before you dive into our Java samples, you should download the sample code from our website. We provide a zipped file that includes the Java code for both samples.

To download the TWS Java API samples

- 1 Click the following link to go to the Application Programming Interfaces page on our website:
http://individuals.interactivebrokers.com/en/p.php?f=programInterface&p=g&ib_entity=lic
- 2 Click the *Proprietary API* tab, then click the *Getting Started Guide* link.
- 3 Scroll down to the Java API section and save the *JavaAPIExamples.zip* file to your computer.
- 4 Extract the files from the zip file to your computer.

What's In the Zipped Sample File?

After you unzip the sample zip file, you'll see the following folders:



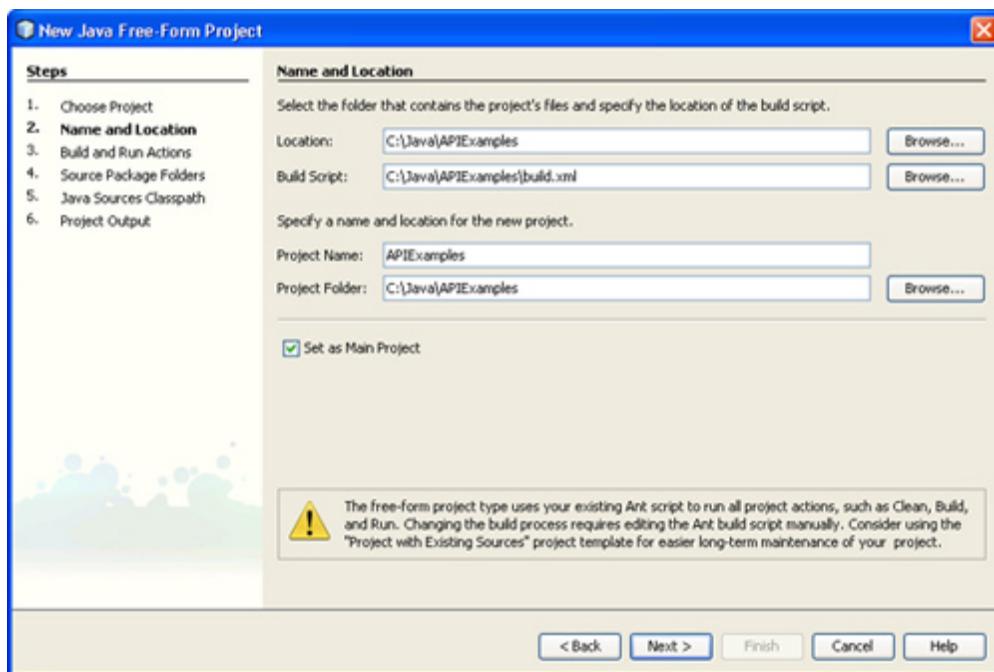
We've included all of the source files you'll need to run the samples, including the TWS Java API. In addition, we've included a build file, *build.xml*, to simplify the build process.

Setting Up the Project in NetBeans

In this section, we're going to show you one way to set up a project for the API samples in NetBeans. We'll set up a new project using NetBeans' Java Free-Form Project option. Of course, you can use any Java IDE to create your java project using our samples.

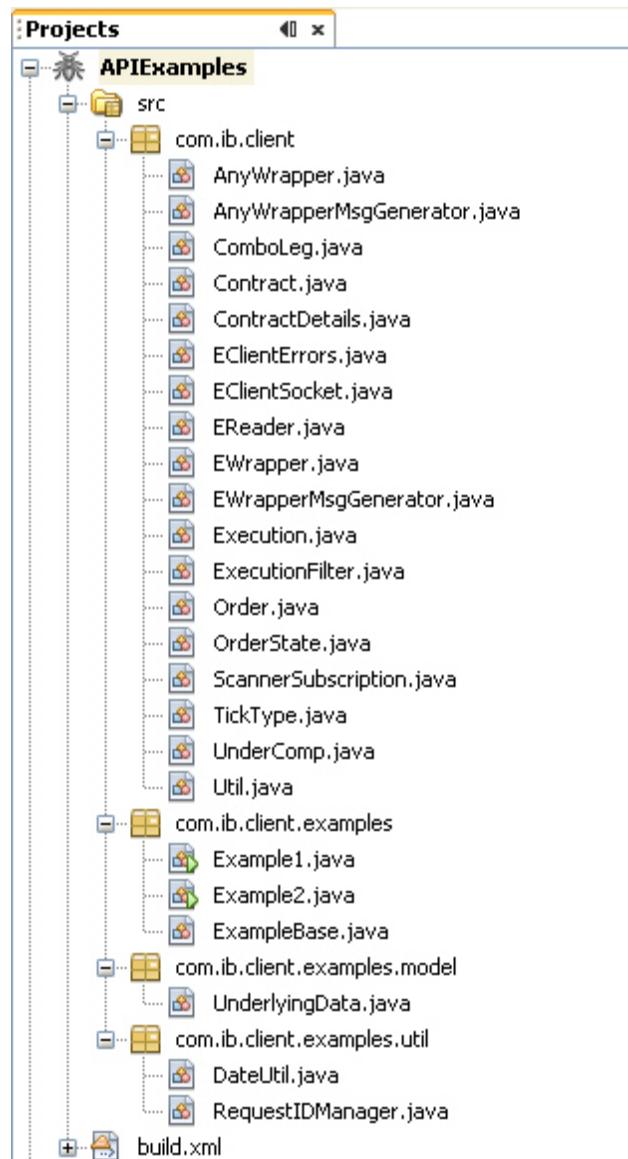
To set up a new project in NetBeans for the samples

- 1 Open NetBeans, then select *New Project* from the **File** menu.
- 2 In the New Project dialog, select *Java Free-Form Project*, then click **Next**.
- 3 In the *Location* field, enter the path to the *APIExamples* folder created when you unzipped the Java API samples file. For example, *C:\APIExamples*.
The Build Script field should be filled automatically with the path to the *build.xml* file. For example, *C:\APIExamples\build.xml*.
- 4 Enter a project name and project folder in the appropriate fields, or accept the default project name folder. *Set as Main Project* is checked by default; you can leave this setting intact if you want, then click **Next**.



- 5 Click **Next** again to display the Source Package folders screen.
- 6 Click **Add Folder**, then browse to the *src* folder in the *APIExamples* folder created when you unzipped the Java API samples file. For example, *C:\APIExamples\src*.
- 7 Click **Finish**.

- 8 The APIExample project should look something like this in the Netbeans Projects window:



A Quick Look at the New Project

In this section, we'll briefly look at the source files in the APIExample project. Again, our descriptions below are based on using Netbeans as the Java IDE, but you are free to use any Java IDE you prefer.

- **com.ib.client** - contains our TWS Java API, including all the methods and parameters and objects you'll need to connect to TWS.
- **com.ib.client.examples** - contains the Example 1.java and Example 2.java classes, as well as ExampleBase.java, which is the base class for both examples. ExampleBase also implements EWrapper.
- **com.ib.client.examples.model** - contains the UnderlyingData.java class, which is used in Example 2.
- **com.ib.client.examples.util** - contains the DateUtil.java class, which is a date utility used in Example 2, and the RequestIDManager.java class, which is an ID-request utility used in Example 2.
- **build.xml** - This is the build file for the project.

Now that you have the APIExamples project created in NetBeans (or your favorite Java IDE), you're ready to take a look at the first sample.

Chapter 22 - Example 1: Requesting Market Data

In this simple sample, we show you how to get the last price for a given symbol.

Run Example 1

Before we take a look at the actual code behind this sample, let's see what happens when we run it. This example does not include any GUI elements; when you run the program from within NetBeans or your favorite Java IDE, it will display in the Output window. To run Example 1 from NetBeans, select `ex1` as your build target.



*Example 1 is designed to run on your local computer "out of the box." If you receive a connection error when you try to run Example 1, open `ExampleBase.java` and change the value for **TWS HOST** = on line 19 from "localhost" to the IP address where TWS is running.*

```
:Output - APIExamples (ex1)
clean:
Deleting directory C:\JavaAPI\APIExamples\build
Created dir: C:\JavaAPI\APIExamples\build
make:
Compiling 24 source files to C:\JavaAPI\APIExamples\build
Building jar: C:\JavaAPI\APIExamples\build\api-examples.jar
Moving 1 file to C:\JavaAPI\APIExamples\lib
ex1:
Server Version:42
TWS Time at connection:20080929 09:39:44 EST
[API.msg2] Market data farm connection is OK:hkfarm {-1, 2104}
[API.msg2] Market data farm connection is OK:usfarm {-1, 2104}
[API.msg2] Market data farm connection is OK:usfuture {-1, 2104}
[API.msg2] Market data farm connection is OK:eurofarm {-1, 2104}
[API.msg2] HMDS data farm connection is OK:ushmds2a {-1, 2106}
[Info] Last price for IBKR was: 23.72
[API.msg3] socket closed
[API.connectionClosed] Closed connection with TWS
BUILD SUCCESSFUL (total time: 7 seconds)|
```

Note: Any stock or option symbols displayed are for illustrative purposes only and are not intended to portray a recommendation.

As you can see from the preceding figure, Example 1 does the following:

- Establishes a connection with TWS.
- Gets the last price for symbol IBKR (this is the symbol for our own Interactive Brokers, but you can change the code to get market data for any symbol).
- Closes the connection with TWS.

Now let's take a look at the code behind this simple example.

What Happens When You Run Example 1?

You've seen Example 1 run, but what's really happening when you run it? Let's take a quick look.

- 1 Example1.java runs.
- 2 Example1 connects to TWS by calling the **connectToTWS()** method defined in ExampleBase.java. The connection parameters are also defined in ExampleBase.
- 3 Example1 creates a contract by calling the **createContract()** method, which is defined in ExampleBase.java with default values. The symbol, IBKR, is defined in the build file, *build.xml*.
- 4 Example1 implements the **tickPrice()** TWS Java API Ewrapper method in order to get the last price for the symbol IBKR (or whatever symbol you define as the command line argument).
- 5 Example1 runs a while loop that checks for the last price every one second, then displays the last price in the output if the time it takes to get the value from TWS is less than a MAX_WAIT_COUNT value, which is set in ExampleBase.
- 6 If it takes more than 15 seconds to get the last price from TWS, the program reports an error, then disconnects from TWS.
- 7 If the program is successful, the last price for the symbol defined as the command line argument is reported in the output, then the program disconnects from TWS.

That's the basic process. Example1 uses the following TWS Java API EClientSocket methods:

- eConnect()
- reqMktData()
- eDisconnect()

and this TWS Java API EWrapper method:

- tickPrice()

Now we're going to get into the details of each step.

Looking at Example1.java

Here's what happens in Example1.java:

- The first thing that Example1.java does when it runs is import the **Contract** and **TickType** SocketClient properties from our Java API. Contract contains attributes used to describe a contract, including symbol, security type, exchange and currency. TickType defines the values for the tickType parameter, which the sample program uses to retrieve the last price of the specified symbol:

```
import com.ib.client.Contract;
import com.ib.client.TickType;
```

- Next, Example1 extends ExampleBase and initializes the *symbol*, *requestID* and *lastPrice* attributes. Note that in our sample code, ExampleBase is a Thread, and therefore Example1 is by inheritance.

```
public class Example1 extends ExampleBase {
    private String symbol = null;
    private int requestId = 0;
    private double lastPrice = 0.0;
```

- The code then sets the Example1 constructor and sets the symbol variable:

```
public Example1(String symbol) {
    this.symbol = symbol;
}
```

- Example1 then declares the main method, which sets a single argument, creates a message that displays only if there is no symbol defined as the command line argument, and calls *start*, which automatically calls the **run()** method because it's in a thread:

```
public static void main(String[] args) {
    if (args.length != 1) {
        System.out.println(" Usage: java Example1 <symbol>");
        System.exit(1);
    } else {
        new Example1(args[0]).start();
    }
}
```

- As described in the preceding bullet, Example1 next calls the **run()** method. This method uses a Java *try* Block to connect to TWS, create a contract, request snapshot market data, runs a while loop that checks for the last price every one second and either reports an error or displays the last price of the specified symbol, and finally disconnects from TWS:

```
public void run() {  
    try {  
        boolean isSuccess = false;  
        int waitCount = 0;  
  
        // Make connection  
        connectToTWS();  
  
        // Create a contract, with defaults...  
        Contract contract = createContract(symbol, "STK", "SMART",  
        "USD");  
  
        // Requests snapshot market data  
        eClientSocket.reqMktData(requestId++, contract, null, true);  
  
        while (!isSuccess && waitCount < MAX_WAIT_COUNT) {  
            // Check if last price loaded  
            if (lastPrice != 0.0) {  
                isSuccess = true;  
            }  
  
            if (!isSuccess) {  
                sleep(WAIT_TIME); // Pause for 1 second  
                waitCount++;  
            }  
        }  
  
        // Display results  
        if (isSuccess) {  
            System.out.println(" [Info] Last price for " + symbol + " was:  
" + lastPrice);  
        } else {  
            System.out.println(" [Error] Failed to retrieve last price  
for " + symbol);  
        }  
    } catch (Throwable t) {  
        System.out.println("Example1.run() :: Problem occurred during  
processing: " + t.getMessage());  
    } finally {  
        disconnectFromTWS();  
    }  
}
```



If you're not sure what a Thread or a try Block is in Java, we strongly recommend that you look these concepts up in your favorite Java programming book or on Sun's Java website before you continue.

- Example1 also implements our **tickPrice()** Ewrapper method, setting the lastPrice variable:

```
public void tickPrice(int tickerId, int field, double price, int
    canAutoExecute) {
    if (field == TickType.LAST) {
        lastPrice = price;
    }
}
```

Now let's dive even deeper into the code by taking a look at what's inside the **run()** method.

Connecting to TWS

The first thing that the **run()** method inside Example1 does is to initiate a *try* Block, which begins by declaring the following local variables:

- *isSuccess* - This is a boolean variable that is initially set to false. You can see this variable in action in the while loop.
- *waitCount* - This integer is initially set to 0. You will also see this variable used in the while loop.

The **run()** method inside Example1 includes these lines of code:

```
// Make connection
connectToTWS();
```

The **connectToTWS()** method, which is NOT part of our TWS Java API, is defined in ExampleBase:

```
public abstract class ExampleBase extends Thread implements EWrapper {
    protected EClientSocket eClientSocket = new EClientSocket(this);
    protected final static String TWS_HOST = "localhost";
    protected final static int TWS_PORT = 7496;
    protected final static int TWS_CLIENT_ID = 1;
    protected final static int MAX_WAIT_COUNT = 15; // 15 secs
    protected final static int WAIT_TIME = 1000; // 1 sec

    protected void connectToTWS() {
        eClientSocket.eConnect(TWS_HOST, TWS_PORT, TWS_CLIENT_ID);
    }
}
```

As you can see from the code, ExampleBase is a Thread that extends the TWS Java API EWrapper, then sets the values of these connection parameters:

- TWS_HOST - This is the host name or IP address of the machine where TWS is running. If you leave this parameter blank, localhost will be used.
- TWS_PORT - This is the port specified in the API Socket Port field in TWS' API Configuration screen

- **TWS_CLIENT_ID** - This is the number TWS uses to identify the client connection. Remember that each client must connect with a unique client ID.

These are the parameters in the eClientSocket **eConnect()** method. ExampleBase also sets the values of MAX_WAIT_COUNT and WAIT_TIME, which we'll see are used in the while loop in Example1.

After setting the values of these parameters, ExampleBase calls the EClientSocket **eConnect()** method, including the three connection parameters.



*Example 1 is designed to run on your local computer "out of the box." If you receive a connection error when you try to run Example 1, change the value for **TWS HOST** = on line 19 in ExampleBase from "localhost" to the IP address where TWS is running.*

Creating a Contract

The next thing that the run() method in Example1 does is to create a contract object containing four parameters:

```
// Create a contract, with defaults...
Contract contract = createContract(symbol, "STK", "SMART", "USD");
```

It creates the contract by calling the **createContract()** overloaded method in ExampleBase. Note that this method is not part of our TWS Java API.

createContract() Method

```

protected Contract createContract(String symbol, String
    securityType, String exchange, String currency) {
    return createContract(symbol, securityType, exchange,
        currency, null, null, 0.0);
}

protected Contract createContract(String symbol, String
    securityType, String exchange, String currency, String expiry,
    String right, double strike) {
    Contract contract = new Contract();

    contract.m_symbol = symbol;
    contract.m_secType = securityType;
    contract.m_exchange = exchange;
    contract.m_currency = currency;

    if (expiry != null) {
        contract.m_expiry = expiry;
    }

    if (strike != 0.0) {
        contract.m_strike = strike;
    }

    if (right != null) {
        contract.m_right = right;
    }

    return contract;
}

```

The parameters included in this method are listed below.

- *symbol* - This variable is initialized in Example1.java, but the value is set in the *build.xml* build file. In our sample, we use IBKR.
- *securityType* - The value for this parameter is set to STK (for stock) in the Example1 call to the **createContract()** method.
- *exchange* - The value for this parameter is set to SMART (for IBSmartRouting) in the Example1 call to the **createContract()** method.
- *currency* - The value for this parameter is set to USD (for US dollars) in the Example1 call to the **createContract()** method.



*This method uses the TWS Java API **Contract** SocketClient property but only makes use of a few of the attributes in that object. For a complete list of all of the attributes in the Contract object, see the [API Reference Guide](#). You could modify the code in our sample to retrieve market data for an option instead a stock by changing*

*the value of the securityType parameter in **createContract()** and using additional Contract attributes such as m_expiry, m_strike and m_right.*

Getting a Snapshot of Market Data

The next task performed by the **run()** method in Example1.java is requesting a snapshot of market data:

```
// Requests snapshot market data
eClientSocket.reqMktData(requestId++, contract, null, true);
```

Notice that this bit of code is calling the **reqMktData()** EClientSocket method, which IS part of our TWS Java API. The parameters are:

- *requestID* - This is a unique value used to identify this market data request. Here we are simply automatically incrementing the request ID numbers. You can also use a specific number for this parameter, such as 1, instead of our little automation.
- *contract* - This is the **Contract** SocketClient object that contains attributes that describe the contract that we've already defined. In our sample, remember that we're looking for market data for IBKR.
- *genericTicklist* - This is set to *null*, which means that we're not using this parameter in this sample program.
- *snapshot* - This boolean parameter is set to *true* here, which tells TWS to return a single snapshot of market data instead of data that dynamically updates. When you use *snapshot* (set the parameter to *true*), you cannot enter any values for *genericTicklist*.

The while Loop

The **run()** method starts out by declaring a pair of local variables, *isSuccess* and *waitCount*. We use this variables in the while loop that is part of the code that gets the last price of the specified contract.

In addition to these variables, the while loop also uses MAX_WAIT_COUNT and WAIT_TIME to evaluate whether or not to display the last price of the specified contract.

```
while (!isSuccess && waitCount < MAX_WAIT_COUNT) {
    // Check if last price loaded
    if (lastPrice != 0.0) {
        isSuccess = true;
    }
    if (!isSuccess) {
        sleep(WAIT_TIME); // Pause for 1 second
        waitCount++;
    }
}
// Display results
if (isSuccess) {
    System.out.println(" [Info] Last price for " + symbol + "
was: " + lastPrice);
} else {
    System.out.println(" [Error] Failed to retrieve last price
for " + symbol);
}
```

The while loop checks for the last price every one second, then displays the last price in the output if the time it takes to get the value from TWS is less than the value of MAX_WAIT_COUNT value, which is set to 15 seconds in ExampleBase. If it takes 15 seconds or less to retrieve the last price of the specified stock, that price is displayed in the output when the program runs. If it takes longer than 15 seconds to get the last price from TWS, the error message indicated in the preceding code snippet is displayed, then the program disconnects from TWS. You can change this wait time to a value longer or shorter than 15 seconds by changing the value of MAX_WAIT_COUNT in ExampleBase.java.

Getting the Last Price

The piece of code that actually sets the last price of the specified contract is the **tickPrice()** EWrapper method, which appears at the very end of Example1.java:

```
public void tickPrice(int tickerId, int field, double price, int
canAutoExecute) {
    if (field == TickType.LAST) {
        lastPrice = price;
    }
}
```

tickPrice() has four parameters:

- *tickerId* - This is the ID that was previously specified in the **reqMktData()** method.
- *field* - This parameter specifies the type of price to retrieve. In our sample, we want to retrieve the LAST price, but we could just as easily specified the bid, ask, high or low, as well as any other type of price that is defined in the TickType object.
- *price* - This parameter holds the price for the price type specified in *field*.

- *canAutoExecute* - This is a boolean parameter that specifies whether the price tick is eligible for automatic execution. A value of 0 (zero) indicates that the the price tick is NOT eligible; a value of 1 indicates that the price tick IS eligible.

The last piece of this code snippet contains an if statement that sets the *lastPrice* attribute (that was initialized at the beginning of Example1.java) to the price parameter in **tickPrice()** if the *field* parameter is set to LAST. **tickPrice()** typically returns a series of prices from TWS: bid, ask, last, and so on. This if statement basically means that as soon the last price is sent from TWS, the *lastPrice* attribute in our sample code is set to that last price from TWS. Our sample is only interested in the last price.

Disconnecting from TWS

The *try* Block ends by disconnecting the sample program from TWS:

```
finally {
    disconnectFromTWS();
```

As you can see, we're calling the **disconnectFromTWS()** method, which is defined in ExampleBase:

```
protected void disconnectFromTWS() {
    if (eClientSocket.isConnected()) {
        eClientSocket.eDisconnect();
```

This code calls the EClientSocket **eDisconnect()** method if the EClientSocket **isConnected()** method is running. **isConnected()** checks to see if there is a connection with TWS; **eDisconnect()** simply disconnects from TWS.



For descriptions of all of the EClientSocket methods in our TWS Java API, see the [API Reference Guide](#).

The build.xml Build File

Last but not least, our example programs include a build file, *build.xml*. It is in this file that we specify the specific contract symbol whose last price we are interested in getting. In our sample, we've set that to IBKR (Interactive Brokers of course), but you can test the program out with any symbol you like. Go ahead and try changing this value on line 40 in the build file.



If you're not familiar with build files, we recommend looking for more information about Ant or on Sun's Java website.

This concludes our discussion of the first of two sample programs that we have provided. In the next chapter, we take a look at the second sample program, which is more complicated than Example 1.

Chapter 23 - Example 2: Automating Option Orders

In this sample, we request market data for a given underlying, then automatically place a buy straddle order for options based on implied volatility and historical volatility.

Run Example 2

Before we take a look at the actual code behind this sample, let's see what happens when we run it. This example does not include any GUI elements; when you run the program from within NetBeans or your favorite Java IDE, it will display in the Output window. To run Example 2 from NetBeans, select ex2 as your build target.



*Example 2 is designed to run on your local computer "out of the box." If you receive a connection error when you try to run Example 2, open ExampleBase.java and change the value for **TWS HOST** = on line 19 from "localhost" to the IP address where TWS is running.*

```
Output - APIExamples (ex2)
Created dir: C:\JavaAPI\APIExamples\build
make:
Compiling 24 source files to C:\JavaAPI\APIExamples\build
Building jar: C:\JavaAPI\APIExamples\build\api-examples.jar
Moving 1 file to C:\JavaAPI\APIExamples\lib
ex2:
Server Version:43
TWS Time at connection:20081003 11:53:24 BST
[API.nextValidId] Next Valid Order ID: 3
[API.msg2] Market data farm connection is OK:hkfarm {-1, 2104}
[API.msg2] Market data farm connection is OK:usfarm {-1, 2104}
[API.msg2] Market data farm connection is OK:usfuture {-1, 2104}
[API.msg2] Market data farm connection is OK:eurofarm {-1, 2104}
[API.msg2] EMDS data farm connection is inactive but should be available upon demand.euhmds2 {-1, 2107}
[API.msg2] EMDS data farm connection is OK:ushmds2a {-1, 2106}
[API.tickPrice] id=0 bidPrice=22.98 canAutoExecute
[API.tickPrice] id=0 askPrice=22.99 canAutoExecute
[API.tickPrice] id=0 lastPrice=22.98 noAutoExecute
[API.tickPrice] id=0 high=23.15 noAutoExecute
[API.tickPrice] id=0 low=22.57 noAutoExecute
[API.tickPrice] id=0 close=22.51 noAutoExecute
[API.tickPrice] id=0 open=22.57 noAutoExecute
[API.tickGeneric] id=0 bidSize=0.8095903764810449
[API.tickGeneric] id=0 bidSize=0.6892248466389005
[API.contractDetailsEnd] reqId = 1 ===== end =====
[Debug] Filtered option contract: IBKR 20081121 22.5 C
[Debug] Filtered option contract: IBKR 20081121 22.5 P
[Info] Buy straddle market order submitted for: UnderlyingData [ symbol: IBKR, minImpVol: 65.0, volRatioLimit: 0.75, lastP]
[API.orderStatus] order status: orderId=3 clientId=1 permId=1192625051 status=Filled filled=1 remaining=0 avgFillPrice=2.7
[API.orderStatus] order status: orderId=3 clientId=1 permId=1192625051 status=Filled filled=1 remaining=0 avgFillPrice=2.7
[API.orderStatus] order status: orderId=4 clientId=1 permId=1192625052 status=Submitted filled=0 remaining=1 avgFillPrice=
[API.orderStatus] order status: orderId=4 clientId=1 permId=1192625052 status=Filled filled=1 remaining=0 avgFillPrice=2.3
[API.orderStatus] order status: orderId=3 clientId=1 permId=1192625051 status=Filled filled=1 remaining=0 avgFillPrice=2.7
[API.orderStatus] order status: orderId=4 clientId=1 permId=1192625052 status=Filled filled=1 remaining=0 avgFillPrice=2.3
```

As you can see from the preceding figure, Example 2 does the following:

- Establishes a connection with TWS.
- Gets the last price, option implied volatility and option historical volatility for IBKR, which is the underlying in this sample.
- Checks the minimum implied volatility and the implied volatility-to-historical volatility ratio of the underlying to see if these values are within specified limits. The values for minimum implied volatility and the implied volatility-to-historical volatility ratio in our example code are specified in the build file, *build.xml*.
- If the values are not within specified limits, displays a message instead of placing the straddle order.
- If the values are within specified limits, gets all available options for the underlying and filters the options to get the one with the price closest to the last price of the underlying and with an expiry of at least 15 days from the current date.
- Places a straddle order for the option that meets the filter criteria.
- Closes the connection with TWS.



In our Example 2, we set the values for minimum implied volatility and the implied volatility-to-historical volatility ratio as command line arguments (in our sample, we've set these values in the build file). In the real world, however, it is up to you to figure out your own calculations on which to base such option order automation.

Now let's take a look at the code behind this example.

What Happens When You Run Example 2?

You've seen Example 2 run, but what's really happening when you run it? Let's take a quick look.

- 1 Example2.java runs.
- 2 Example2 connects to TWS by calling the **connectToTWS()** method defined in ExampleBase.java. The connection parameters are also defined in ExampleBase.
- 3 Example2 calls the **retrieveUnderlyingData()** method defined in Example2.java, which creates a contract by calling the **createContract()** method defined in ExampleBase.java with default values, then calls the TWS Java API EClientSocket method **reqMktData()** to request market data, using RequestIDManager.java to keep track of request IDs. The symbol, IBKR, is a command line argument.
- 4 Example2 implements the **tickPrice()** TWS Java API Ewrapper method to get the last price for the symbol IBKR (or whatever symbol you specify) from TWS, and the **tickGeneric()** TWS Java API Ewrapper method to get the option implied volatility and option historical volatility for the symbol.
- 5 **retrieveUnderlyingData()** runs a while loop that checks for the last price, option implied volatility and option historical volatility every one second, then displays the prices in the output if the time it takes to get the value from TWS is less than a MAX_WAIT_COUNT value, which is set in ExampleBase.

- 6 If it takes more than 15 seconds to get the three values from TWS, the program reports an error, then disconnects from TWS. If the program is successful, it stores the prices in UnderlyingData.java.
- 7 UnderlyingData.java implements two methods, **isDataReady()**, which checks to see if the price data has been set, and **isOrderCriteria()**, which checks the data to see if meets the program's criteria for placing a straddle order. This initial criteria is based on two additional values, minimum implied volatility (*minImpVol*) and *volRatioLimit*, which are command line arguments.
- 8 If the criteria is not met, a message is displayed in the output and the program disconnects from TWS.
- 9 If the criteria is met, Example2 runs the **retrieveOptionContracts()** method, which finds all available option contracts for the underlying symbol.
- 10 Once all the option contracts have been found, Example2 runs the **filterContracts()** method, which finds all options with underlying prices as close to the original underlying's last price as possible. **filterContracts()** then finds the option with the expiry date that is at least 15 days from today (the date the program is run).
- 11 RequestIDManager.java checks to make sure the next order ID is ready, then a straddle order is placed using the TWS Java API **placeOrder()** method, actually placing a market buy order for a call option and a second market buy order for a put option.
- 12 The program finally waits for TWS Java API Ewrapper method **orderStatus()** to return the order status to the output, then disconnects from TWS.

That's the basic process. Example2 uses the following TWS Java API EClientSocket methods:

- eConnect()
- reqMktData()
- cancelMktData()
- reqContractDetails()
- eDisconnect()
- placeOrder()

and the following TWS Java API EWrapper methods:

- tickPrice()
- tickGeneric()
- contractDetails()
- contractDetailsEnd()
- orderStatus()
- nextValidId()

Now lets go into the details of each step.

Looking at Example2.java

Here's what happens in Example2.java:

- The first thing that Example2.java does when it runs is import several SocketClient properties from our Java API: Contract, Contract Details, Order, TickType. Contract contains attributes used to describe a contract, including symbol, security type, exchange and currency. Contract Details contains additional attributes that describe a contract, including valid order types and exchanges, and bond details. Order contains attributes that describe an order. TickType defines the values for the tickType parameter, which the sample program uses to retrieve the last price of the specified symbol. Example2 also imports EWrapperMsgGenerator, which creates messages for a variety of data. Example2.java also imports the UnderlyingData, DateUtil and RequestIDManager, which are included in the additional sources for the examples:

```
import com.ib.client.Contract;
import com.ib.client.Contract;
import com.ib.client.ContractDetails;
import com.ib.client.EWrapperMsgGenerator;
import com.ib.client.Order;
import com.ib.client.TickType;
import com.ib.client.examples.model.UnderlyingData;
import com.ib.client.examples.util.DateUtil;
import com.ib.client.examples.util.RequestIDManager;
import java.util.ArrayList;
import java.util.ArrayList;
import java.util.List;
```

- Next, Example2 extends ExampleBase and initializes GENERIC_TICKS, UnderlyingData and a list of option contracts. Note that in our sample code, ExampleBase is a Thread, and so Example2 is one by inheritance.

```
public class Example2 extends ExampleBase {

    private final static String GENERIC_TICKS = "104, 106"; // Hist
    Vol, Imp Vol
    private UnderlyingData underlyingData = null;
    private List<Contract> optionContracts = new ArrayList<Contract>();
```

- The code then sets the Example2 constructor and sets the underlyingData variable.

```
public Example2(String symbol, String minImpVol, String volRatioLimit)
{
    this.underlyingData = new UnderlyingData(symbol,
        Double.parseDouble(minImpVol), Double.parseDouble(volRatioLimit));
}
```

- Example2 then declares the main method, which sets three arguments, creates a message that displays only if there is no symbol, minImpVol and volRatioLimit defined, and calls start, which automatically calls the **run()** method because it's a Thread:

```
public static void main(String[] args) {  
    if (args.length != 3) {  
        System.out.println(" Usage: java Example2 <symbol> <minImpVol>  
                           <volRatioLimit>");  
        System.exit(1);  
    } else {  
        new Example2(args[0], args[1], args[2]).start();  
    }  
}
```

- As described in the preceding bullet, Example2 next calls the **run()** method. This method connects to TWS and calls **retrieveUnderlyingData()**, which creates a contract, requests market data, runs a while loop that checks for the three prices every one second and reports an error or displays the prices, then checks the data to see if the price data has been set and if it meets the program criteria for placing a straddle order. If the criteria is not met, a message is displayed in the output and the program disconnects from TWS. If the criteria is met, **run()** calls the **retrieveOptionContracts()** method, which finds all available option contracts for the underlying symbol and then, once all the option contracts have been found, runs the **filterContracts()** method, which finds all options with underlying prices as close to the original underlying's last price as possible, and the option with the expiry date that is at least 15 days from today (the date the program is run). RequestIDManager.java checks to make sure the next order ID is ready, then **run()** places a straddle order using the TWS Java API **placeOrder()** method, actually placing a market buy order for a call option and a second market buy order for a put option. The program finally calls TWS Java API Ewrapper method **orderStatus()** to return the order status to the output, then disconnects from TWS.

The **run()** method code is shown on the next page.

```
public void run() {
    try {
        connectToTWS();

        // Retrieve underlying data (last price, hist vol, imp vol)
        if (retrieveUnderlyingData()) {
            if (underlyingData.isOrderCriteriaMet()) {
                // Retrieve option contracts for underlying
                if (retrieveOptionContracts()) {
                    // Find the one that is 1) closest strike to underlying last
                    // price, 2) expiry not within 15 days
                    Contract callContract = filterContracts("C");
                    Contract putContract = filterContracts("P");

                    if (RequestIDManager.singleton().isOrderIdInitialized()) {
                        // Place buy straddle for 1 contract as market order
                        Order callOrder = createOrder("BUY", 1, "MKT");
                        Order putOrder = createOrder("BUY", 1, "MKT");

                        eClientSocket.placeOrder(RequestIDManager.singleton() .
                            getNextOrderId(), callContract, callOrder);
                        eClientSocket.placeOrder(RequestIDManager.singleton() .
                            getNextOrderId(), putContract, putOrder);
                        System.out.println(" [Info] Buy straddle market order
                            submitted for: " + underlyingData.toString());
                        sleep(WAIT_TIME * 30); // Hang around for order status updates...
                    } else {
                        System.out.println(" [Error] Failed to initialize order ID
                            for: " + underlyingData.toString());
                    }
                } else {
                    System.out.println(" [Error] Failed to retrieve option contracts
                        for: " + underlyingData.toString());
                }
            } else {
                System.out.println(" [Info] Underlying does NOT meet order
                    criteria:" + underlyingData.toString());
            }
        } else {
            System.out.println(" [Error] Failed to retrieve underlying data:
                " + underlyingData.toString());
        }
    } catch (Throwable t) {
        System.out.println(" [Error] Example1.run() :: Problem occurred
            during processing: " + t.getMessage());
    } finally {
        disconnectFromTWS();
    }
}
```



If you're not sure what a Thread is in Java, we strongly recommend that you look these concepts up in your favorite Java programming book or on Sun's Java website before you continue.

- Example2 contains the following methods, which are unique to our sample code:
 - **retrieveUnderlyingData()**
 - **retrieveOptionContracts()**
 - **filterContracts()**
- Example2 also implements the following TWS Java API EWrapper methods:
tickPrice()

```
public void tickPrice(int tickerId, int field, double price, int canAutoExecute) {
    System.out.println(" [API.tickPrice] " +
EWrapperMsgGenerator.tickPrice(tickerId, field, price,
canAutoExecute));

    if (field == TickType.LAST) {
        underlyingData.setLastPrice(price);
    }
}
```

tickGeneric()

```
public void tickGeneric(int tickerId, int field, double generic) {
    System.out.println(" [API.tickGeneric] " +
EWrapperMsgGenerator.tickGeneric(tickerId, tickerId, generic));

    if (field == TickType.OPTION_IMPLIED_VOL) {
        underlyingData.setImpVol(generic * 100);
    } else if (field == TickType.OPTION_HISTORICAL_VOL) {
        underlyingData.setHistVol(generic * 100);
    }
}
```

contractDetails()

```
public void contractDetails(int reqId, ContractDetails contractDetails) {
    // System.out.println(" [API.contractDetails] " +
EWrapperMsgGenerator.contractDetails(reqId, contractDetails));

    if (contractDetails != null && contractDetails.m_summary !=
null && "OPT".equals(contractDetails.m_summary.m_secType)) {
        optionContracts.add(contractDetails.m_summary);
    }
}
```

contractDetailsEnd()

```

public void contractDetailsEnd(int reqId) {
    System.out.println(" [API.contractDetailsEnd] " +
EWrapperMsgGenerator.contractDetailsEnd(reqId));

    RequestIDManager.singleton().addToRequestsCompleted(reqId);
}

```

orderStatus()

```

public void orderStatus(int orderId, String status, int filled, int
remaining, double avgFillPrice, int permId, int parentId, double
lastFillPrice, int clientId, String whyHeld) {
    System.out.println(" [API.orderStatus] " +
EWrapperMsgGenerator.orderStatus(orderId, status, filled, remaining,
avgFillPrice, permId, parentId, lastFillPrice, clientId, whyHeld));
}

```

nextValidId()

```

public void nextValidId(int orderId) {
    System.out.println(" [API.nextValidId] " +
EWrapperMsgGenerator.nextValidId(orderId));
    RequestIDManager.singleton().initializeOrderId(orderId);
}
}

```

Now we'll look at the details of the **run()** method.

Connecting to TWS

The first thing that the **run()** method inside Example2 does is to initiate a *try* Block, which begins by calling the **connectToTWS()** method:

```

// Make connection
connectToTWS();

```

The **connectToTWS()** method, which is NOT part of our TWS Java API, is defined in ExampleBase:

```

public abstract class ExampleBase extends Thread implements EWrapper {
    protected EClientSocket eClientSocket = new EClientSocket(this);
    protected final static String TWS_HOST = "localhost";
    protected final static int TWS_PORT = 7496;
    protected final static int TWS_CLIENT_ID = 1;
    protected final static int MAX_WAIT_COUNT = 15; // 15 secs
    protected final static int WAIT_TIME = 1000; // 1 sec

    protected void connectToTWS() {
        eClientSocket.eConnect(TWS_HOST, TWS_PORT, TWS_CLIENT_ID);
    }
}

```

As you can see from the code, ExampleBase is a thread that extends the TWS Java API EWrapper, then sets the values of these connection parameters:

- TWS_HOST - This is the host name or IP address of the machine where TWS is running. If you leave this parameter blank, localhost will be used.
- TWS_PORT - This is the port specified in the API Socket Port field in TWS' API Configuration screen
- TWS_CLIENT_ID - This is the number TWS uses to identify the client connection. Remember that each client must connect with a unique client ID.

These are the parameters in the eClientSocket **eConnect()** method. ExampleBase also sets the values of MAX_WAIT_COUNT and WAIT_TIME, which we'll see are used in the while loop in Example1.

After setting the values of these parameters, ExampleBase calls the EClientSocket **eConnect()** method, including the three connection parameters.



*Example 2 is designed to run on your local computer "out of the box." If you receive a connection error when you try to run Example 1, change the value for **TWS HOST** = on line 19 in ExampleBase from "localhost" to the IP address where TWS is running.*

Retrieving the Underlying Data

The next thing that the **run()** method must do is retrieve market data for the specified underlying, IBKR, then retrieve option contracts for the underlying if certain criteria are met. It does this by running a series of if statements, which call the **retrieveUnderlyingData()** method and then, if the criteria are met, calls the **retrieveOptionContracts()** method to further filter the results and place a straddle order.

Let's look at **retrieveUnderlyingData()**, which is defined in Example2.java:

```

private boolean retrieveUnderlyingData() throws InterruptedException {
    boolean isSuccess = false;
    int waitCount = 0;

    // Create a contract, with defaults...
    Contract contract = createContract(underlyingData.getSymbol(), "STK",
        "SMART", "USD");

    // Requests market data
    int requestId = RequestIDManager.singleton().getNextRequestId();
    eClientSocket.reqMktData(requestId, contract, GENERIC_TICKS, false);

    while (!isSuccess && waitCount < MAX_WAIT_COUNT) {
        // Check if last price and volatilities loaded
        if (underlyingData.isDataReady()) {
            isSuccess = true;
        } else {
            sleep(WAIT_TIME); // Pause for 1 sec
            waitCount++;
        }
    }

    // Cancel market data
    eClientSocket.cancelMktData(requestId);

    return isSuccess;
}

```

retrieveUnderlyingData() begins by declaring the following local variables:

- *isSuccess* - This is a boolean variable that is initially set to False. You can see this variable in action in the while loop.
- *waitCount* - This integer is initially set to 0. You will also see this variable used in the while loop.

Creating a Contract

The next thing that the **retrieveUnderlyingData()** does is to create a contract object containing four parameters:

```
protected Contract createContract(String symbol, String
    securityType, String exchange, String currency) {
    return createContract(symbol, securityType, exchange,
        currency, null, null, 0.0);
}

protected Contract createContract(String symbol, String
    securityType, String exchange, String currency, String expiry,
    String right, double strike) {
    Contract contract = new Contract();

    contract.m_symbol = symbol;
    contract.m_secType = securityType;
    contract.m_exchange = exchange;
    contract.m_currency = currency;

    if (expiry != null) {
        contract.m_expiry = expiry;
    }

    if (strike != 0.0) {
        contract.m_strike = strike;
    }

    if (right != null) {
        contract.m_right = right;
    }

    return contract;
}
```

It creates the contract by calling the **createContract()** method in ExampleBase. Note that both **retrieveUnderlyingData()** and **createContract()** are not part of our TWS Java API.

createContract() Method

```
protected Contract createContract(String symbol, String
    securityType, String exchange, String currency) {
    return createContract(symbol, securityType, exchange,
        currency, null, null, 0.0);
}
```

The parameters included in this method are listed below.

- *symbol* - This variable is initialized in Example2.java, but the value is set in the *build.xml* build file. In our sample, we use IBKR.
- *securityType* - The value for this parameter is set to STK (for stock) in the Example1 call to the **createContract()** method.

- *exchange* - The value for this parameter is set to SMART (for IBSmartRouting) in the Example1 call to the **createContract()** method.
- *currency* - The value for this parameter is set to USD (for US dollars) in the Example1 call to the **createContract()** method.



*This method uses the TWS Java API **Contract** SocketClient property but only makes use of a few of the attributes in that object. For a complete list of all of the attributes in the Contract object, see the [API Reference Guide](#). You could modify the code in our sample to retrieve market data for an option instead a stock by changing the value of the securityType parameter in **createContract()** and using additional Contract attributes such as m_expiry, m_strike and m_right.*

Requesting Market Data

Next, **retrieveUnderlyingData()** requests market data, using the code in RequestIDManager.java to keep track of request IDs. The symbol, IBKR, is a command line argument.

```
/// Requests market data
int requestId = RequestIDManager.singleton().getNextRequestId();
eClientSocket.reqMktData(requestId, contract, GENERIC_TICKS, false);
```

RequestIDManager is our own little piece of code that manages all of the market data request IDs and order IDs so that you don't have to.

Notice that this bit of code is calling the **reqMktData()** EClientSocket method, which is part of our TWS Java API. The parameters are:

- *requestID* - This is a unique value used to identify this market data request. Here we are simply automatically incrementing the request ID numbers. You can also use a specific number for this parameter, such as 1, instead of our little automation.
- *contract* - This is the **Contract** SocketClient object that contains attributes that describe the contract that we've already defined. In our sample, remember that we're looking for market data for IBKR.
- *GENERIC_TICKS* - This was initialized at the beginning of Example2.java (line 41 if you're keeping track of line numbers) and set to tick values 104 and 106, which will retrieve values for historical volatility and option implied volatility, respectively.



You can see the complete list of tick values in our [API Reference Guide](#), which is available on our website.

- *snapshot* - This boolean parameter is set to *false* here, which tells TWS to market data that dynamically updates. When you use *snapshot* (set the parameter to *true*), you cannot enter any values for *genericTicklist* (or in this case, *GENERIC_TICKS*, which we initialized at the beginning of Example2.java).

The while Loop

Recall that **retrieveUnderlyingData()** started out by declaring a pair of local variables, *isSuccess* and *waitCount*. Just as in Example 1, we use these variables in the while loop in **retrieveUnderlyingData()** that is part of the code that gets the market data of the specified contract and checks to see if the values for last price, option implied volatility and historical volatility are set.

The **isDataReady()** method is responsible for performing this check:

isDataReady() Method

```
public boolean isDataReady() {
    if (symbol == null || minImpVol == 0.0 || volRatioLimit == 0.0 ||
        lastPrice == 0.0 || impVol == 0.0 || histVol == 0.0) {
        return false;
    } else {
        return true;
    }
}
```

In addition, the while loop also uses MAX_WAIT_COUNT and WAIT_TIME to evaluate whether or not to display the three prices of the specified contract.

```
while (!isSuccess && waitCount < MAX_WAIT_COUNT) {
    // Check if last price and volatilities loaded
    if (underlyingData.isDataReady()) {
        isSuccess = true;
    } else {
        sleep(WAIT_TIME); // Pause for 1 sec
        waitCount++;
    }
}
```

The while loop checks for the three prices every one second, then displays the prices in the output if the time it takes to get the value from TWS is less than the value of MAX_WAIT_COUNT value, which is set to 15 seconds in ExampleBase. If it takes 15 seconds or less to retrieve the prices of the specified stock, those prices are displayed in the output when the program runs. If it takes longer than 15 seconds to get the prices from TWS, the error message indicated in the preceding code snippet is displayed, then the program disconnects from TWS. You can change this wait time to a value longer or shorter than 15 seconds by changing the value of MAX_WAIT_COUNT in ExampleBase.java.

Getting the Last Price, Option Implied Volatility and Historical Volatility

The code that actually returns the last price is the **tickPrice()** EWrapper method, and the code that actually returns the option implied volatility and historical volatility is the **tickGeneric()** EWrapper method, which are implemented in Example2.java:

tickPrice() Method

```
public void tickPrice(int tickerId, int field, double price, int
    canAutoExecute) {
    System.out.println(" [API.tickPrice] " +
        EWrapperMsgGenerator.tickPrice(tickerId, field, price,
            canAutoExecute));

    if (field == TickType.LAST) {
        underlyingData.setLastPrice(price);
    }
}
```

tickPrice() has four parameters:

- *tickerId* - This is the ID that was previously specified in the **reqMktData()** method.
- *field* - This parameter specifies the type of price to retrieve. In our sample, we want to retrieve the LAST price, but we could just as easily specified the bid, ask, high or low, as well as any other type of price that is defined in the **TickType** object.
- *price* - This parameter holds the price for the price type specified in *field*.
- *canAutoExecute* - This is a boolean parameter that specifies whether the price tick is eligible for automatic execution. A value of 0 (zero) indicates that the the price tick is NOT eligible; a value of 1 indicates that the price tick IS eligible.

The last piece of this code snippet contains an if statement that sets the **lastPrice** attribute (that was initialized at the beginning of Example1.java) to the **price** parameter in **tickPrice()** if the *field* parameter is set to LAST. **tickPrice()** typically returns a series of prices from TWS: bid, ask, last, and so on. This if statement basically means that as soon the last price is sent from TWS, the **lastPrice** attribute in our sample code is set to that last price from TWS. Our sample is only interested in the last price.

tickGeneric() Method

```
public void tickGeneric(int tickerId, int field, double generic) {
    System.out.println(" [API.tickGeneric] " +
        EWrapperMsgGenerator.tickGeneric(tickerId, field, generic));

    if (field == TickType.OPTION_IMPLIED_VOL) {
        underlyingData.setImpVol(generic * 100);
    } else if (field == TickType.OPTION_HISTORICAL_VOL) {
        underlyingData.setHistVol(generic * 100);
    }
}
```

tickGeneric() has three parameters:

- *tickerId* - This is the ID that was previously specified in the **reqMktData()** method.
- *field* - This parameter specifies the type of price to retrieve. In our sample, we want to retrieve the option implied volatility and historical volatility.
- *canAutoExecute* - This is a boolean parameter that specifies whether the price tick is eligible for automatic execution. A value of 0 (zero) indicates that the the price tick is NOT eligible; a value of 1 indicates that the price tick IS eligible.

The last piece of this code snippet contains an if statement that basically means that as soon the option implied volatility and historical volatility are sent from TWS, the corresponding attributes in our sample code are set to those values from TWS.

Retrieving Options Contracts

When the values for last price, option implied volatility and historical volatility are received, the **run()** method runs **isOrderCriteriaMet()**, which checks to see if the initial criteria for placing a straddle order have been met by the data and is defined in UnderlyingData.java.

isOrderCriteriaMet() Method

```
public boolean isOrderCriteriaMet() {  
    if (isDataReady() && impVol >= minImpVol && (impVol /  
    histVol > volRatioLimit)) {  
        return true;  
    } else {  
        return false;  
    }  
}
```

Recall that if this initial criteria is met, then we retrieve all of the available option contracts for the underlying. The initial criteria are:

- The three data values must be set (**isDataReady()**)
- AND *impVol* (option implied volatility) is equal to or greater than the *minImpVol* (set to 125 in our build file)
- AND option implied volatility divided by historical volatility is greater than *volRatioLimit* (set to 2 in our build file).

If all three of these criteria are met by the retrieved market data, then and only then will the program retrieve all the available option contracts for the underlying (IBKR).

If the criteria is NOT met, a message is displayed in the output and the program disconnects from TWS. If the criteria IS met, the next if statement in the **run()** method retrieves all the available option contracts for the underlying.

The **run()** method calls the **retrieveOptionContracts()** method to get all of the available option contracts for the underlying.

retrieveOptionContracts() Method

```
private boolean retrieveOptionContracts() throws InterruptedException {
    boolean isSuccess = false;
    int waitCount = 0;

    // Find all option contracts for underlying, will filter strike and
    // expiry later...
    Contract contract = createContract(underlyingData.getSymbol(), "OPT",
        "SMART", "USD");

    int requestId = RequestIDManager.singleton().getNextRequestId();
    eClientSocket.reqContractDetails(requestId, contract);

    while (!isSuccess && waitCount < MAX_WAIT_COUNT) {
        // Check if all contracts received
        if (RequestIDManager.singleton().isRequestComplete(requestId)) {
            isSuccess = true;
        } else {
            sleep(WAIT_TIME); // Pause for 1 sec
            waitCount++;
        }
    }
    return isSuccess;
}
```

This method creates a contract for an option using SMART as the exchange and USD as the currency, and uses the RequestIDManager code to manage the market data request IDs.

retrieveOptionContracts() then requests contract details for all available option contracts for the underlying by calling the **reqContractDetails()** TWS Java API EClientSocket method. Finally it runs a while loop that does pretty much the same thing as the previous while loop we saw - it checks for the last option contract every one second, and times out if the time it takes to get the next contract from TWS is more than the value of MAX_WAIT_COUNT value, which is set to 15 seconds in ExampleBase. You can change this wait time to a value longer or shorter than 15 seconds by changing the value of MAX_WAIT_COUNT in ExampleBase.java.

contractDetails() and contractDetailsEnd()

The TWS Java API Ewrapper method **contractDetails()** returns the details from TWS for the available option contracts. After all of the available option contract details have been received, the Ewrapper method **contractDetailsEnd()** is received from TWS. **contractDetailsEnd()** serves as an end marker for the contract details and tells our little sample program that all of the contract details have been received.

Filtering the Option Contracts

Before the program can place a straddle order however, it must filter the available option contracts that it retrieved. It runs the **filterContracts()** method in Example2.java, which uses two *for* loops to filter this data for two criteria:

- It finds the option contracts whose strike price is closest to the last price of the underlying, and then
- It finds the option contract whose expiry is at least 15 days away from the current date (today). The program uses DateUtil.java to do this.

filterContracts() Method

```
private Contract filterContracts(String right) {  
    Contract c = null;  
  
    // First by price  
    double priceDiff = 0.0;  
  
    for (Contract contract : optionContracts) {  
        if (contract.m_right.equals(right)) {  
            if (c == null) {  
                c = contract;  
                priceDiff = Math.abs(contract.m_strike - underlyingData.getLastPrice());  
            } else {  
                double tempDiff = Math.abs(contract.m_strike - underlyingData.getLastPrice());  
                if (tempDiff < priceDiff) {  
                    c = contract;  
                    priceDiff = tempDiff;  
                }  
            }  
        }  
    }  
  
    // Next find closest expiry outside 15 days  
    long days = 0;  
  
    for (Contract contract : optionContracts) {  
        // This time include check to look at those matching strike  
        if (contract.m_right.equals(right) && contract.m_strike == c.m_strike) {  
            if (days == 0) {  
                days = DateUtil.getDeltaDays(contract.m_expiry);  
                c = contract;  
            } else {  
                long tempDays = DateUtil.getDeltaDays(contract.m_expiry);  
                if (tempDays < days && tempDays > 15) {  
                    days = tempDays;  
                    c = contract;  
                }  
            }  
        }  
    }  
  
    System.out.println(" [Debug] Filtered option contract: " + c.m_symbol + " " + c.m_expiry  
        + " " + c.m_strike + " " + c.m_right);  
  
    return c;  
}
```

If neither of these filtering processes finds an option contract, the straddle order is not placed and messages are displayed in the output before the program disconnects from TWS.

Placing the Straddle Order

If an option contract IS found; that is, its strike price is closest to the last price of the underlying, and its expiry is at least 15 days away from the current date; then **run()** places a buy straddle order for a call and a put. It calls the TWS Java API **placeOrder()** EClientSocket method to create two orders for one option each, one a market buy for a put, the other a market buy for a call. Here again is the code that handles the orders:

```
if (RequestIdManager.singleton().isOrderIdInitialized()) {
    // Place buy straddle for 1 contract as market order
    Order callOrder = createOrder("BUY", 1, "MKT");
    Order putOrder = createOrder("BUY", 1, "MKT");

    eClientSocket.placeOrder(RequestIDManager.singleton().getNextOrderId(),
        callContract, callOrder);
    eClientSocket.placeOrder(RequestIDManager.singleton().getNextOrderId(),
        putContract, putOrder);
    System.out.println(" [Info] Buy straddle market order submitted for: " +
        underlyingData.toString());
    sleep(WAIT_TIME * 30); // Hang around for order status updates...
} else {
```

The EWrapper method **orderStatus()** is called to report the status of the orders in the output.

Disconnecting from TWS

The **run()** method ends by disconnecting the sample program from TWS:

```
finally {
    disconnectFromTWS();
```

As you can see, we're calling the **disconnectFromTWS()** method, which is defined in **ExampleBase**:

```
protected void disconnectFromTWS() {
    if (eClientSocket.isConnected()) {
        eClientSocket.eDisconnect();
```

This code calls the EClientSocket **eDisconnect()** method if the EClientSocket **isConnected()** method is running. **isConnected()** checks to see if there is a connection with TWS; **eDisconnect()** simply disconnects from TWS.



For descriptions of all of the EClientSocket methods in our TWS Java API, see the [API Reference Guide](#).

The build.xml Build File

Example 2 uses the same build file, *build.xml*, as Example 1. It is in this file that we specify the specific contract symbol whose last price we are interested in getting, as well as the option implied volatility and historical volatility (the command line arguments). In our sample, we've set these to IBKR (Interactive Brokers of course), 125 and 2, but you can test the program out with any symbol or trading strategy you prefer.



If you're not familiar with build files, we recommend looking for more information about Ant or on Sun's Java website.

This concludes the discussion of our two code samples. The next section of this book tells you where you can get more information about our TWS Java API.

Where to Go from Here

If you've come this far and actually read the book, you now have a pretty decent grasp on what the Java API can do, and how to make it do some of the things you want. Now we give you a bit more information about how to link to TWS with our Java API, and we suggest some helpful outside resources you can use to help you move forward.

This section contains the following chapters:

- [Chapter 24 - Linking to TWS using the TWS Java API](#)
- [Chapter 25 - Additional Resources](#)

Chapter 24 - Linking to TWS using the TWS Java API

If you have the skill and confidence to handle Java on your own, you can build your own Java application to link to TWS, using the following steps as a guide.

- 1 Import **com.ib.client.*** into your source code file. This is the package that contains the TWS Java API classes and methods.
- 2 Implement the **EWrapper** interface. This class will receive messages from the socket.
- 3 Override the following methods:

EWrapper Method	Description
tickPrice()	Handles market data.
tickSize()	
tickOptionComputation()	
tickGeneric()	
tickString()	
tickEFP()	
orderStatus()	Receives order status.
openOrder()	Receives open orders.
error()	Receives error information.
connectionClosed()	Notifies you when TWS terminates the connection.
updateAccountValue()	Receives current account values.
updateAccountTime()	Receives the last time account information was updated.
updatePortfolio()	Receives current portfolio information.
nextValidId()	Receives the next valid order ID upon connection.
contractDetails()	Receives contract information.
contractDetailsEnd()	Identifies the end of a given contract details request.
bondContractDetails()	Receives bond contract information.
execDetails()	Receives execution report information.
updateMktDepth()	Receives market depth information.
updateMktDepthL2()	Receives Level II market depth information.
updateNewsBulletin()	Receives IB news bulletins.

EWrapper Method	Description
managedAccounts()	Receives a list of Financial Advisor (FA) managed accounts.
receiveFA()	Receives FA configuration information.
historicalData()	Receives historical data results.
scannerParameters()	Receives an XML document that describes the valid parameters of a scanner subscription.
scannerData()	Receives market scanner results.
scannerDataEnd()	Called when the scanner snapshot is received and marks the end of one scan.
realTimeBar()	Receives real-time bars.
currentTime()	Receives the current system time on the server.
fundamentalData()	Receives Reuters global fundamental market data.

- 4** Instantiate the **EClientSocket** class. This object will be used to send messages to TWS.
- 5** Call the following EClientSocket methods:

EClientSocket Method	Description
eConnect()	Connects to TWS.
eDisconnect()	Disconnects from TWS.
reqMktData()	Requests market data.
cancelMktData()	Cancels market data.
reqMktDepth()	Requests market depth.
cancelMktDepth()	Cancels market depth.
reqContractDetails()	Requests contract details.
placeOrder()	Places an order.
cancelOrder()	Cancels an order.
reqAccountUpdates()	Requests account values, portfolio, and account update time information.
reqExecutions()	Requests a list of the day's execution reports.
reqOpenOrders()	Requests a list of current open orders for the requesting client and associates TWS open orders with the client. The association only occurs if the requesting client has a Client ID of 0.
reqAllOpenOrders()	Requests a list of all open orders.

EClientSocket Method	Description
reqAutoOpenOrders()	Automatically associates a new TWS with the client. The association only occurs if the requesting client has a Client ID of 0.
reqNewsBulletin()	Requests IB news bulletins.
cancelNewsBulletins()	Cancels IB news bulletins.
setServerLogLevel()	Sets the level of API request and processing logging.
reqManagedAccts()	Requests a list of Financial Advisor (FA) managed account codes.
requestFA()	Requests FA configuration information from TWS.
replaceFA()	Modifies FA configuration information from the API.
reqScannerParameters()	Requests an XML document that describes the valid parameters of a scanner subscription.
reqScannerSubscription()	Requests market scanner results.
cancelScannerSubscription()	Cancels a scanner subscription.
reqHistoricalData()	Requests historical data.
cancelHistoricalData()	Cancels historical data.
reqRealTimeBars()	Requests real-time bars.
cancelRealTimeBars()	Cancels real-time bars.
exerciseOptions()	Exercises options.
reqCurrentTime()	Requests the current server time.
serverVersion()	Returns the version of the TWS instance to which the API application is connected.
TwsConnectionTime()	Returns the time the API application made a connection to TWS.
reqFundamentalData()	Requests Reuters global fundamental data. There must be a subscription to Reuters Fundamental set up in Account Management before you can receive this data.
cancelFundamentalData()	Cancels Reuters global fundamental data.

Chapter 25 - Additional Resources

There are many resources out there that will be adequate in getting you where you need to go. If you have some books or places that you like, feel free to stick with them. The following are the resources we find most helpful, and perhaps they'll be good to you, too!

Help with Java Programming

While this book is intended for users with Java programming experience, we understand that even experienced Java programmers need help every once in a while.

The best place to go to find additional help with all things Java is the Sun web site. Just type <http://java.sun.com> in your browser's address line and check out the list of links under *Resources* on the right side of the page. Sun has many online resources available for Java programmers, including documentation, tutorials, and code samples.

If you simply want to look up information about the actual Java API (as opposed to our TWS Java API), you can go directly to Sun's [API Specifications Reference page](#). There you will find links to documentation, Javadocs, technical articles and a whole host of useful information.

There are literally hundreds of additional printed and web-based resources for Java programmers. We encourage you to investigate these on your own.

Help with the Java API

For help specific to the Java TWS API, the one best place to go, really the ONLY place to go, is the Interactive Brokers website. Once you get there, you have lots of resources. Just type www.interactivebrokers.com in your browser's address line. Now that you're there, let me tell you where you can go.



As of this writing in March 2011, the IB website looks as I'm describing. IB has a tendency to revamp the look and organization of their site every year or two, so have a little patience if it looks slightly different from what's described here.

The API Reference Guide

The API Reference Guide includes sections for each API technology, including the DDE for Excel. The upper level topics which are shown directly below the main book are applicable across the board to all or multiple platforms.

To access the API Reference Guide from the IB web site, select *API Solutions* from the **Trading** menu, then click the **IB API** button, then click the **Reference Guide** tab. Click the **Online API Reference Guide** button to open the online guide, which contains a section devoted entirely to the DDE for Excel API.

The API Beta and API Production Release Notes

The beta notes are in a single page file, and include descriptions of any new additions to the API (all platforms) that haven't yet been pushed to production. The API Release Notes opens an index page that includes links to all of the past years' release notes pages. The index provides one-line titles of all the features included in each release.

To access these notes from the IB web site, select *API Solutions* from the **Trading** menu, then click the **IB API** button, then click the **Release Notes** tab and select a link to the latest API production release notes. You can also access the release notes for the latest API Beta release from this page.

The TWS API Webinars

IB hosts free online webinars through WebEx to help educate their customers and other traders about the IB offerings. They present the API webinar about once per month, and have it recorded on the website for anyone to listen to at any time.

- To register for the API webinar, from the IB web site click **Education**, then select **Webinars**. Click the **Live Webinars** button, then click the **API** tab.
- To view the recorded version of the API webinar, from the **Live Webinars** page click the **Watch Previously Recorded Webinars** button. Links to recorded versions of previously recorded webinars are listed on the page.

API Customer Forums

You can trade ideas and send out pleas for help via the IB customer base accessible through both the IB Bulletin Board and the Traders' Chat. The bulletin board includes a thread for the API, and thus provides an ongoing transcript of questions and answers in which you might find the answer to your question. The Traders' Chat is an instant-message type of medium and doesn't retain any record of conversations.

- "To view or participate in the IB Bulletin Board, go to the **Education** menu and click *Bulletin Boards & Chats*. Click the **Bulletin Board** tab, then click the **Launch IB Discussion Forum button** to access all of our bulletin boards, including the TWS API bulletin board.
- To participate in the Traders' Chat, you need to click the **Chat** icon from the menu bar on TWS. Note that both of these customer forums are for IB customers only.

IB Customer Service

IB customers can also call or email customer service if you can't find the answer to your question. However, IB makes it clear that the APIs are designed for use by programmers and that their support in this area is limited. Still, the customer service crew is very knowledgeable and will do their best to help resolve your issue. Simply send an email to:

api@interactivebrokers.com

IB Features Poll

The IB Features Poll lets IB customers submit suggestions for future product features, and vote and comment on existing suggestions.

From the IB web site, click **About IB**, then select *New Features Poll*. Suggestions are listed by category; click a plus sign next to a category to view all feature suggestions for that category. To submit a suggestion, click the *Submit Suggestion* link.

Appendix A - Extended Order Attributes

Attribute	Valid Values
timeInForce	DAY GTC OPG IOC
ocaGroup	String
account	The account number, used for institutional and advisor accounts.
open/close	O, C (for institutions)
origin	0, 1 (for institutions)
orderRef	String
transmit	0 (don't transmit) 1 (transmit)
Parent order Id	String (the order ID used for the parent order, use for bracket and auto trailing stop orders)
blockOrder	0 (not a block order) 1 (this is a block order)
sweepToFill	0 (not a sweep-to-fill order) 1 (this is a sweep-to-fill order)
displaySize	String (publicly disclosed order size)

Appendix A - Extended Order Attributes

Attribute	Valid Values
triggerMethod	<p>Specifies how simulated Stop, Stop-Limit, and Trailing Stop orders are triggered.</p> <p>0 - the default value. The "double bid/ask" method will be used for orders for OTC stocks and US options. All other orders will use the "last" method.</p> <p>1 - use "double bid/ask" method, where stop orders are triggered based on two consecutive bid or ask prices.</p> <p>2 - "last" method, where stop orders are triggered based on the last price.</p> <p>3 - "double-last" method, where stop orders are triggered based on last two prices.</p> <p>4 - "bid-ask" method. For a buy order, a single occurrence of the bid price must be at or above the trigger price. For a sell order, a single occurrence of the ask price must be at or below the trigger price.</p> <p>7 - "last-or-bid-ask" method. For a buy order, a single bid price or the last price must be at or above the trigger price. For a sell order, a single ask price or the last price must be at or below the trigger price.</p> <p>8 - "mid-point" method, where the midpoint must be at or above (for a buy) or at or below (for a sell) the trigger price, and the spread between the bid and ask must be less than 0.1% of the midpoint.</p>
Hidden	<p>Only valid for orders routed to Island.</p> <p>0 - False</p> <p>1 (order not visible when viewing market depth)</p>
Discretionary Amount	Used in conjunction with a limit order to give the order a greater price range over which to execute.
Good After Time	Enter the date and time after which the order will become active. Use the format YYYYMMDD hh:mm:ss
Good 'Till Date	The order continues working until the close of market on the date you enter. Use the format YYYYMMDD. To specify a time of day to close the order, enter the time using the format HH:MM:SS. Specify the time zone using a valid three-letter acronym.
FA Group	For Advisor accounts only. The name of the Account Group.
FA Method	<p>For Advisor accounts only. The share allocation method.</p> <p>EqualQuantity</p> <p>NetLiq</p> <p>AvailableEquity</p> <p>PctChange</p>
FA Percentage	For Advisor accounts only. The share allocation percentage.
FA Profile	For Advisor accounts only. The name of the Share Allocation profile.
Short Sale Slot	<p>For institutional accounts only; for SSHORT actions.</p> <p>1 – If you hold the shares</p> <p>2 – Shares will be delivered from elsewhere.</p>
Short Sale Location	If shares are delivered from elsewhere, enter where in a comma-delimited list with no spaces. For institutional accounts only.

Attribute	Valid Values
OCA Type	1 = Cancel on Fill with Block 2 = Reduce on Fill with Block 3 = Reduce on Fill without Block
Rule 80A	Individual = 'I' Agency = 'A', AgentOtherMember = 'W' IndividualPTIA = 'J' AgencyPTIA = 'U' AgentOtherMemberPTIA = 'M' IndividualPT = 'K' AgencyPT = 'Y' AgentOtherMemberPT = 'N'
Settling Firm	Institutions only
All or None	0 = false 1 = true
Minimum Qty	Identifies the order as a minimum quantity order.
Percent Offset	The percent offset for relative orders.
Electronic Trade Only	0 = false 1 = true
Firm Quote Only	0 = false 1 = true
NBBO Price Cap	Maximum SMART order distance from the NBBO.
Auction Strategy	For BOX exchange only. match = 1 improvement = 2 transparent = 3
Starting Price	The starting price. For BOX orders only.
Stock Ref Price	Used for VOL orders to compute the limit price sent to an exchange (whether or not Continuous Update is used), and for price range monitoring. Also used for price improvement option orders.
Delta	The stock delta. For BOX orders only.
Stock Range Lower	The lower value for the acceptable underlying stock price range. For price improvement option orders on BOX and VOL orders with dynamic management.
Stock Range Upper	The upper value for the acceptable underlying stock price range. For price improvement option orders on BOX and VOL orders with dynamic management.
Volatility	The option price in volatility, as calculated by TWS ' Option Analytics. This value is expressed as a percent and is used to calculate the limit price sent to the exchange.

Attribute	Valid Values
Volatility Type	1 = daily 2 = annual
Reference Price Type	1 = average 2 = BidOrAsk
Hedge Delta Order Type	Prior to TWS Release 859, use "1" to send a market order, "0" for no order. After TWS 859, enter an accepted order type such as: MKT, LMT, REL.
Continuous Update	0 = false 1 = true
Hedge Delta Aux Price	Enter the Aux Price for Hedge Delta order types that require one.
Trail Stop Price	Used for Trailing Stop Limit orders only. This is the stop trigger price for TRAILLIMIT orders.
Scale Num Components	Used for Scale orders only, this value defines the number of components in the order.
Scale Component Size	NO LONGER SUPPORTED
Scale Price Increment	Used for Scale orders only, this value is used to calculate the per-unit price of each component in the order. This cannot be a negative number.
Outside RTH	0 = false 1 = true

Appendix B - Account Page Values

Field	Description	Notes
Account Code	The account number.	
Account Type	Identifies the IB account type.	
Accrued Cash	Reflects the current month's accrued debit and credit interest to date, updated daily.	At the beginning of each month, the past month's accrual is added to the cash balance and this field is zeroed out.
Available Funds	For securities: Equity with Loan Value - Initial margin For commodities: Net Liquidation Value - Initial margin	
Buying Power	Cash Account : (Minimum (Equity with Loan Value, Previous Day Equity with Loan Value)- Initial Margin) Standard Margin Account : Available Funds*4	
Cash Balance	For securities: Settled cash + sales at the time of trade For commodities: Settled cash + sales at the time of trade + futures PNL	
Currency	Shows the currency types that are listed in the Market Value area.	
Cushion	Shows your current margin cushion.	
Day Trades Remaining	Number of day trades left for pattern day trader period.	
Day Trades Remaining T+1, T+2, T+3, T+4	The number of day trades you have left for a 4-day pattern day-trader.	

Appendix B - Account Page Values

Field	Description	Notes
Equity With Loan Value	<p>For Securities:</p> <ul style="list-style-type: none"> • Cash Account: Settled Cash • Margin Account: • Total cash value + stock value + bond value + (non-U.S. & Canada securities options value) <p>For Commodities:</p> <ul style="list-style-type: none"> • Cash Account: Total cash value + commodities option value - futures maintenance margin requirement + minimum (0, futures PNL) • Margin Account: Total cash value + commodities option value - futures maintenance margin requirement 	
Excess Liquidity	Equity with Loan Value - Maintenance margin	
Exchange Rate	The exchange rate of the currency to your base currency.	
Full Available Funds	<p>For securities:</p> <p>Equity with Loan Value - Initial margin</p> <p>For commodities:</p> <p>Net Liquidation Value - Initial margin</p>	
Full Excess Liquidity	Equity with Loan Value - Maintenance margin	
Full Init Margin Req	Overnight initial margin requirement in the base currency of the account.	
Full Maint Margin Req	Maintenance margin requirement as of next period's margin change in the base currency of the account.	
Future Option Value	Real-time mark-to-market value of futures options.	
Futures PNL	Real-time change in futures value since last settlement.	
Gross Position Value	Long Stock Value + Short Stock Value + Long Option Value + Short Option Value.	
Init Margin Req	Initial margin requirement in the base currency of the account.	

Field	Description	Notes
Leverage	<p>For Securities:</p> <ul style="list-style-type: none"> • Gross Position value / Net Liquidation value <p>For Commodities:</p> <ul style="list-style-type: none"> • Net Liquidation value - Initial margin 	
Look Ahead Available Funds	<p>For Securities:</p> <ul style="list-style-type: none"> • Equity with loan value - look ahead initial margin. <p>For Commodities:</p> <ul style="list-style-type: none"> • Net Liquidation value - look ahead initial margin. 	
Look Ahead Excess Liquidity	Equity with loan value - look ahead maintenance margin.	
Look Ahead Init Margin Req	Initial margin requirement as of next period's margin change in the base currency of the account.	
Look Ahead Maint Margin Req	Maintenance margin requirement as of next period's margin change in the base currency of the account.	
Look Ahead Next Change	Indicates when the next margin period begins.	
Maint Margin Req	Maintenance margin requirement in the base currency of the account.	
Net Liquidation	<p>For Securities:</p> <ul style="list-style-type: none"> • Total cash value + stock value + securities options value + bond value <p>For Commodities:</p> <ul style="list-style-type: none"> • Total cash value + commodities options value 	
Net Liquidation by Currency	Same as above for individual currencies.	
Option Market Value	Real-time mark-to-market value of securities options.	
PNL	The difference between the current market value of your open positions and the average cost, or Value - Average Cost.	
Previous Day Equity with Loan Value	Marginable Equity with Loan Value as of 16:00 ET the previous day, only applicable to securities.	

Appendix B - Account Page Values

Field	Description	Notes
Realized PnL	Shows your profit on closed positions, which is the difference between your entry execution cost and exit execution cost, or (execution price + commissions to open the positions) - (execution price + commissions to close the position).	
Reg T Equity	Initial margin requirements calculated under US Regulation T rules.	
Reg T Margin	<p>For Securities:</p> <ul style="list-style-type: none"> • Cash Account: Settled Cash • Margin Account: Total cash value + stock value + bond value + (non-U.S. & Canada securities options value) <p>For Commodities:</p> <ul style="list-style-type: none"> • Cash Account: Total cash value + commodities option value - futures maintenance margin requirement + minimum (0, futures PNL) • Margin Account: Total cash value - futures maintenance margin requirement 	
SMA	Max ((EWL - US initial margin requirements)*, (Prior Day SMA +/- change in day's cash +/- US initial margin requirements**) for trades made during the day.) *calculated end of day under US Stock rules, regardless of country of trading. **at the time of the trade	Only applicable for securities.
Stock Market Value	Real-time mark-to-market value of stock	
Total Cash Balance	Cash recognized at the time of trade + futures PNL	
Total Cash Value	Total cash value of stock, commodities and securities	

Index

A

account page values B-131
ActionListener for the Cancel Hist. Data button 4-51
ActionListener for the Cancel Mkt Data button 4-45
ActionListener for the Cancel Mkt Depth button 4-48
ActionListener for the Cancel Real Time Bars button 4-54
ActionListener for the Cancel Subscription button 4-58
ActionListener for the Connect button 4-35
ActionListener for the Disconnect button 4-37
ActionListener for the Historical Data button 4-50
ActionListener for the Market Scanner button 4-56
ActionListener for the Req Current Time button 6-80
ActionListener for the Req Mkt Data button 4-39
ActionListener for the Req Mkt Depth button 4-47
ActionListener for the Req Real Time Bars button 4-53
additional resources 8-125
API
 reasons for using 2-17
API beta notes 8-125
API Reference Guide 8-125
API release notes 8-125
API software
 downloading 3-23
 installing 3-25
API support email 8-126
API webinars 8-126

B

bAutoBind 5-76
build.xml 7-101, 7-120

C

Cancel Hist. Data button 4-51
Cancel Mkt Data button 4-45
Cancel Mkt Depth button 4-48
Cancel Real Time Bars button 4-54
canceling a market scanner subscription 4-58
canceling historical data 4-49, 4-51

canceling market data 4-38, 4-45
canceling market depth 4-46, 4-48
canceling market scanner subscriptions 4-55
canceling news bulletins 6-83
canceling orders 5-62, 5-66
canceling real time bars 4-52, 4-54
cancelNewsBulletins() 6-83
cancelOrder() 5-66
cancelScannerSubscription() 4-58
changing the server logging level 6-84
Class definition of SampleFrame.java 4-34
Client ID and multiple API sessions 5-73
Connect button 4-34
Connect dialog 4-35
connecting the Java Test client to TWS 4-32
connecting to TWS 4-34
contract data 4-59
contract data fields 4-59
contractDetails() 4-60
copyExtendedOrderDetails() 5-72
createButtonPanel() method 4-34
createContract() in Example1 7-98
current time 6-80
currentTime() 6-80
customer forums 8-126
customer service 8-126

D

DDE for Excel API
 additional resources 8-125
 preparing to use 3-21
Disconnect button 4-37
disconnecting from TWS 4-37
document conventions 1-11
downloading API software 3-23
downloading the sample code 7-88

E

EClientSocket constructor 1-11, 4-34
Example 1
 connecting to TWS 7-96
 creating a contract 7-97
 getting a snapshot of market data 7-99
 getting the last price 7-100, 7-115
 running 7-92

while loop 7-99
Example 2
 filtering option contracts 7-118
 isDataReady() 7-114
 isOrderCriteriaMet() 7-116
 placing a straddle order 7-119
 retrieveOptionContracts() 7-111
 7
 retrieveUnderlyingData() 7-111
 retrieving market data 7-113
 running 7-102
 while loop in
 retrieveUnderlyingData() 7-114
Example1
 disconnecting 7-101
Example1.java 7-94
Example2
 connecting 7-109
 contractDetails() 7-108
 creating a contract 7-112
 nextValidId() 7-109
 orderStatus() 7-109
 retrieving underlying data 7-110
 run() method 7-107
 tickGeneric() 7-108
 tickPrice() 7-108
Example2.java 7-105
examples
 build.xml 7-101, 7-120
 Example 1 - Requesting Market Data 7-92
 Example 2 - Automating Option Orders 7-102
 preparing the sample code 7-88
 setting up a project in NetBeans 7-89
execDetails() 5-78
execDetails() parameters 5-78
Execution Filter dialog 5-77
execution information display 5-78
executions 5-77
Exercise Options button 5-68
exercise options fields 5-69
exerciseOptions() 5-70
exerciseOptions() parameters 5-70
exercising options 5-68
Extended button 5-71
Extended order attributes 5-71
Extended Order Attributes dialog 5-71
extended order attributes A-127

F

Features Poll 8-126
footnotes and references 1-9
framework of Java Test Client 4-32

H

historical data 4-49
Historical Data button 4-50
historical data fields 4-49
historicalData() 4-51
how to use this book 1-8

I

IB bulletin boards 8-126
IB Customer Service 8-126
icons used in this book 1-10
installing API software 3-25
introduction 1-7, 4-31, 5-61, 6-79

J

J2SE Development Kit and NetBeans IDE Bundle 3-22
Java API, help with 8-125
Java IDE, downloading 3-22
Java JDK, downloading 3-22
Java programming help 8-125
Java Test Client
 connecting to TWS 4-32
 framework 4-32
Java Test Client main window 4-32

L

Log Configuration dialog 6-85
log.txt file 6-84
logLevel 6-85

M

market data 4-38
 canceling 4-45
 EWrapper methods 4-44
 snapshot 4-45
market data returned 4-43
market depth 4-46
market depth fields 4-46
Market Scanner button 4-56
market scanners 4-55
modifying orders 5-66
multiple API sessions 5-73

N

NetBeans
 using with sample code 7-89
Netbeans 3-22
News Bulletin Subscription dialog 6-81
news bulletins 6-81

O

onCancelHistoricalData() 4-51
onCancelMktData() 4-45

onCancelMktDepth() 4-48
onCancelOrder() 5-66
onCancelRealTimeBars() 4-54
onCancelSubscription() 4-58
onConnect() method 4-35
onExerciseOptions() 5-68
onExtendedOrder() 5-72
onHistoricalData() 4-50
onPlaceOrder() 5-64
onReqAllOpenOrders() 5-75
onReqAutoOpenOrders() 5-76
onReqContractData() 4-60
onReqCurrentTime() 6-80
onReqExecutions() 5-77
onReqMktData() method 4-39
onReqMktDepth() 4-47
onReqNewsBulletins() 6-81
onReqOpenOrders() 5-74
onReqRealTimeBars() 4-53
onScanner() 4-56
onServerLogging() 6-84
open orders 5-73
openOrder() 5-74
options 5-68
order fields 5-64
orders 5-62
 modifying 5-66
 what-if 5-67
orderStatus() 5-66, 5-74
organization of this book 1-8

P

Place Order button 5-63
placeOrder() 5-65
placeOrder() parameters 5-65
placing orders 5-62
preparing to use the DDE for Excel API 3-21

R

real time bars 4-52
 default bar size 4-53
real-time account monitoring, in TWS 2-16
realtimeBar() 4-54
reasons for using an API 2-17
Req All Open Orders button 5-75
Req Auto Open Orders button 5-76
Req Contract Data button 4-59
Req Current Time button 6-80
Req Executions button 5-77
Req Mkt Depth button 4-47
Req Mkt Data button 4-39
Req News Bulletins button 6-81
Req Open Orders button 5-74
Req Real Time Bars button 4-53
reqAllOpenOrders() 5-75
reqAutoOpenOrders() 5-76
reqContractDetails() 4-60
reqCurrentTime() 6-80
reqExecutions() 5-78

reqHistoricalData() 4-50
reqMktData() 4-41
reqMktData() parameters 4-41
reqMktDepth() 4-47
reqMktDepth() parameters 4-47
reqNewsBulletins() 6-82
reqOpenOrders() 5-74
reqRealTimeBars() 4-53
reqScannerParameters() 4-57
reqScannerSubscription() 4-57
Request All Open Orders 5-74
Request Auto Open Orders 5-74
Request Open Orders 5-74
Request Parameters button 4-56
requesting contract data 4-59
requesting executions 5-77
requesting historical data 4-49
requesting market data 4-38, 4-39
requesting market depth 4-46
requesting market scanner parameters 4-56
requesting open orders 5-73
requesting real time bars 4-52
resources, for Java programming help 8-125
run() method in Example2 7-107
running Example 1 7-92
running Example 2 7-102

S

Sample dialog 4-38, 4-40
contract data fields 4-59
exercise options fields 5-69
historical data fields 4-49
market data fields 4-42
market depth fields 4-46
order fields 5-64
real time bars fields 4-52
SampleFrame.java 4-33
samples 7-87
Scanner dialog 4-55
scannerData() 4-57
scannerDataEnd() 4-57
server log levels 6-85
Server Logging button 6-84
server logging level 6-84

setServerLogLevel() 6-85
snapshot 4-45
subscribing to market scanner subscriptions 4-55
subscribing to news bulletins 6-81

T

tickEFP() 4-44
tickGeneric() 4-44
use in Example 2 7-115
tickOptionComputation() 4-44
tickPrice() 4-44
tickSize() 4-44
tickString() 4-44
Trader Workstation
overview 2-14
trading window 2-15
TWS
real-time account monitoring in 2-16
TWS and the API 2-17
TWS Order Ticket 2-15
TWS overview 2-14, 2-15
TWS Quote Monitor 2-15

U

underlying data
retrieving in Example 2 7-110
updateMktDepth() 4-48
updateMktDepthL2() 4-48
updateNewsBulletin() 6-82
updateNewsBulletin() parameters 6-82
using this book 1-8
document conventions 1-11
icons 1-10
organization 1-8

V

viewing the server logging level 6-84

W

What If button 5-67
what-if data 5-67
whatIf() parameter 5-67