Praise for

Inside the Microsoft Build Engine: Using MSBuild and Team Foundation Build, Second Edition

"Inside the Microsoft Build Engine: Using MSBuild and Team Foundation Build is a practical book covering all the essentials of MSBuild and the Team Foundation Server build system. But what makes the book extra valuable is its focus on real-life scenarios that often are hard to find a good, working solution for. In fact there is information in the book you're unlikely to find anywhere else. With the second edition of the book, the authors fill the gaps again, this time by covering the new TFS build workflow technology as well as MSBuild 4.0. It is an invaluable book that saves lots of time whenever you work with any aspect of automated builds in Visual Studio and TFS. This is a book I'll make sure to have with me all the time!"
-Mathias Olausson, ALM Consultant, QWise/Callista, Sweden

"As an ALM Consultant I come across many teams that are struggling with their build tools and processes. The second edition of Sayed and William's book is the perfect answer for these teams. Not only will it show you how to get your builds back on track, I challenge anyone not to be able to use the information in this book to improve their existing builds. It includes updated content focusing on the new Visual Studio 2010 release and is packed with practical examples you could start using straight away. You simply must include it in your technical library."
-Anthony Borton, Microsoft Visual Studio ALM MVP, Senior ALM trainer/consultant, Enhance ALM Pty Ltd, Australia

"The first edition of Inside the Microsoft Build Engine was a brilliant look at the internals of MSBuild, so it's fantastic to see Sayed and William updating it with all the new features in MSBuild 4.0 and also delving into the Team Foundation Server 2010 workflow based build process. It's also a real pleasure to see deployment with MSDeploy covered so that you can learn not only how to automate your builds, but also how to automate your deployments. A great book. Go out and get a copy now."
-Richard Banks, Visual Studio ALM MVP and Principal Consultant with Readify, Australia

"Did you know about the TaskFactory in MSBuild? If not, you're not alone - but you will know after reading this book. This book provides insights into the current technologies of the Microsoft Build Engine. Starting with background information about MSBuild, it covers also the necessary basics of Workflow Foundation which are applied during the description of advanced topics of Team Foundation Build. The level of detail is targeted to experienced build masters having a development background - even the overview is stuffed with new information, references, hints and best practices about MSBuild. Samples are provided as step-by-step guidance easy to follow inside Visual Studio. What I found astonishing is the practical focus of the samples such as web project deployment. I could have used at least half of them in my development projects! Simply put: A must read for all build experts that have to deal with MSBuild and the Team Foundation Server build engine who are not only interested in solutions but also background information!"
-Sven Hubert, AIT TeamSystemPro Team, Consultant, MVP Visual Studio ALM – www.tfsblog.de

"The reason that I only own one MSBuild/Team Build book is because there is no need for another. This book covers both topics from soup to nuts and is written in a way that allows new users to ramp up quickly. The real-world code examples used to illustrate the topics are useful in their own right. The Second Edition covers all of the changes in MSBuild 4.0 and all of the newness that is Team Build 2010. This is my 'go to' guide, and the only book on these topics that I recommend to my clients."
-Steve St Jean, Visual Studio ALM MVP, DevProcess (ALM) Consultant with Notion Solutions, an Imaginet Company

"Whether you consider yourself experienced or you are taking your first steps in the build and automation arena, this 2nd edition will prove a valuable read. Skilled MSBuild users will do well to remind themselves of the intricacies of MSBuild and learn of the new 4.0 features whilst novices are taken on a steady paced journey to quickly acquire the knowledge and confidence in developing successful solutions. This edition brings additional value to our ever changing profession in discussing MSDeploy and the new Windows Workflow 4.0 based Team Foundation Build. Regardless of your experience, I wholeheartedly recommend this book."
-Mike Fourie, Visual Studio ALM MVP and ALM Ranger, United Kingdom

"The first edition of this book had a perfect balance between a tutorial and a reference book. I say this as I used the book first to kick start my MS Build knowledge and then as reference whenever I needed information on some advanced topic. My main interest is Team Foundation Server and I learned MS Build more from necessity than an urge, hence I was very curious to see the 2nd edition. Sayed and William did not disappoint me - the four chapters on Team Build cover all points needed to customize builds. As a bonus there are three whole chapters on web deployment which is a recurrent request I hear during my consulting and presentations on TFS. If I had to summarize my opinion in a single sentence, I would just say `Buy the book, you won't regret it'."
-Tiago Pascoal, Visual Studio ALM MVP and Visual Studio ALM Ranger, Portugal

"Reliable and repeatable build processes are often the Achilles' heel of development teams. Often this is down to a lack of understanding of the underlying technologies and how they fit together. No matter which Continuous Integration (CI) tool you may be using, this book provides the fundamental information you need to establish solid build and deployment engineering practices and demystifies the various Microsoft technologies used along the way. This book is the essential reference for any team building software on the Microsoft.NET platform."
-Stuart Preston, Visual Studio ALM Ranger and Chief Technology Officer at RippleRock

"Successfully deploying application is one of the big challenges in today's modern software development. As applications become more complex to develop, they also become more complex to deploy. This well-written book provides us a deep-dive on how developers can improve their productivity and accomplish the business needs using Microsoft deployment technology: MSBuild, Web Deploy and Team Build. Microsoft provides us the right tools, and this book provides us the information we need to extract real value from these tools."
-Daniel Oliveira, MVP, Visual Studio ALM Ranger and ALM Consultant at TechResult

Microsoft

Foreword by Brian Harry
Technical Fellow, Team Foundation Server, Microsoft Corp.

Inside the Microsoft® Build Engine

2
SECOND EDITION

Using MSBuild and Team Foundation Build

Sayed Ibrahim Hashimi
William Bartholomew

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Control Number: 2010940848
ISBN: 978-0-7356-4524-0

Printed and bound in the United States of America.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office or contact Microsoft Press International directly at fax (425) 936-7329. Visit our Web site at www.microsoft.com/mspress. Send comments to mspinput@microsoft.com.

Acquisitions Editor: Devon Musgrave
Developmental Editor: Devon Musgrave
Project Editor: Iram Nawaz
Editorial Production: S4Carlisle Publishing Services
Technical Reviewer: Marc H. Young
Cover: Tom Draper Design

Body Part No. X17-29997

I would like to dedicate this book to my parents, Sayed A. Hashimi and Sohayla Hashimi, as well as my college advisor, Dr. Ben Lok. My parents have, over the course of the years, sacrificed a lot to give us the opportunity for us to be able to achieve our dreams. I can only hope that they are proud of the person that I have become. When I first met Ben, I wanted to get into a research program that he had going. Thankfully, he was willing to accept me. Ben helped show me how rewarding hard work can be, and he has enabled me to succeed in my career. When I look back on influences in my life, who are not relatives, he ranks at the top of my list. I am sure that I wouldn't be where I am had it not been for him.

—Sayed Ibrahim Hashimi

To my mother, Rosanna O'Sullivan, and my father, Roy Bartholomew, for their unfaltering support in all my endeavors.

—William Bartholomew

I would like to dedicate this book to my parents, Syama Mohana Rao Adharapurapu and Nalini Adharapurapu, my brother, Raghavendra Adharapurapu, my sister, Raga Sudha Vijjapurapu, and my wife, Deepti Ramakrishna.

—Pavan Adharapurapu

I dedicate this book to my wife, Samantha, and my daughters, Amelie and Madeline, as well as my parents, Leonea and Craig. Their love has no boundaries and their support has made me believe that I can accomplish anything.

—Jason Ward

Contents at a Glance

Table of Contents

Part III Advanced MSBuild Topics

Part IV MSBuild Cookbook

What do you think of this book? We want to hear from you!

Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

microsoft.com/learning/booksurvey

Foreword

Often when people think about build, they think just about the act of compiling some source code – when I hit F5 in the IDE, it builds, right? Well yes, kind of. In a real production build system, there is so much more to it than that. There are many kinds of builds – F5, desktop, nightly, continuous, rolling, gated, buddy etc. The variety of build types is reflective of the important role build plays in the software development process and the varied ways it does so. Build is a key integration point in the process. It is where developers' work comes together; it is where developers hand off to test and where release hands off to operations. No wonder there are so many requirements on it.

As I mentioned, build is about a lot more than compiling the code. It can include making sure the right code is assembled, compiling, testing, version stamping, packaging, deployment and more. Of course, because software systems are all different and organizations are different, many of the activities need to be completely different. As a result, extensibility plays a major role. In TFS 2010, we increased the extensibility options by including a build workflow engine (based on the .NET Workflow Foundation) on top of the existing msbuild capabilities. Unfortunately, as flexibility increases, so does the amount you need to know to make sound decisions and fully automate your build workflow.

This book is a great resource to help you understand the variety of roles build plays in software development and how you can leverage msbuild and TFS. It will show you how to use "out of the box" solutions, provide guidance on when to customize, what the best customization approaches are and details on and examples of how to actually do it. I think it will be an invaluable resource to keep on your reference shelf.

Brian Harry

Technical Fellow

Team Foundation Server, Microsoft

Introduction

Build has historically been kind of like a black art, in the sense that there are just a few people who know and understand build, and are passionate about it. But in today's evolving environment that is changing. Now more and more people are becoming interested in build, and making it a part of their routine development activities. Today's applications are different from those that we were building five to ten years ago. Along with that the process by which we write software is different as well. Nowadays it is not uncommon for a project to have sophisticated build processes which include such things as code generation, code analysis, unit testing, automated deployment, etc. To deal with these changes developers are no longer shielded from the build process. Developers have to understand the build process so that they can leverage it to meet their needs.

Back in 2005 Microsoft released MSBuild, which is the build engine used to build most Visual Studio projects. That release was MSBuild 2.0. Since that release Microsoft has released two major versions of MSBuild—MSBuild 3.5 and MSBuild 4.0. In MSBuild 3.5 Microsoft released such goodness as multi-processor support, multi-targeting, items and properties being defined inside of targets and a few other things which brought MSBuild to where it needed to be. In MSBuild 4.0 there were a lot of really great features delivered. The feature which stands out the most is the support for building Visual C++ projects. Starting with Visual Studio 2010 your Visual C++ project files are in MSBuild format. Modifying MSBuild to be able to support building Visual C++ projects was a big effort on Microsoft's part, but they understood that the value they were delivering to customers would be worth it. Along with support for Visual C++ there were a number of significant feature add ons, such as support for BeforeTargets/AfterTargets, inline tasks, property functions, item functions and a new object model to name a few. During that same period Team Build has undergone a number of big changes.

Team Foundation Build (or Team Build as it is more commonly known) is now in its third version. Team Build 2005 and 2008 were entirely based on MSBuild using it for both build orchestration as well as the build process itself. While this had the advantage of just needing to learn one technology MSBuild wasn't suited for tasks such as distributing builds across multiple machines and performing complex branching logic. Team Build 2010 leverages the formidable combination of Workflow Foundation (for build orchestration) and MSBuild (for build processes) to provide a powerful, enterprise-capable, build automation tool. Team Build 2010 provides a custom Workflow Foundation service host that runs on the build servers that allows the build process to be distributed across multiple machines. The Workflow Foundation based process template can perform any complex branching and custom logic that is supported by Workflow Foundation, including the ability to call MSBuild based project files.

A common companion to build is deployment. In many cases the same script which builds your application is used to deploy it. This is why in this updated book we have a section, Part VII Web Deployment Tool, in which we dedicate three chapters to the topic. MSDeploy is a tool which was first released in 2009. It can be used to deploy websites, and other applications, to local and remote servers. In this section we will show you how to leverage MSDeploy and the Web Publishing Pipeline (WPP) in order to deploy your web applications. Two chapters are devoted to the theory of both MSDeploy and the WPP. There is also a cookbook chapter which shows real world examples of how to use these new technologies. Once you've automated your build and deployment process for the first time you will wonder why you didn't do that for all of your projects.

Who This Book Is For

This book is written for anyone who uses, or is interested in using, MSBuild or Team Build. If you are using Visual Studio to your applications then you are already using MSBuild. *Inside the Microsoft Build Engine* is for all developers and build masters using Microsoft technologies. If you are interested in learning more about how your applications are being built and how you can customize this process then you need this book. If you are using Team Build, or thinking of using it tomorrow, then this book is a must read. It will save you countless hours.

This book will help the needs of enterprise teams as well as individuals. You should be familiar with creating applications using Visual Studio. You are not required to be familiar with the build process, as this book will start from the basics and build on that. Because one of the most effective methods for learning is through examples, this book contains many examples.

Assumptions

To get the most from this book, you should meet the following profile:

- You should be an familiar with Visual Studio
- You should have experience with the technologies you are interested in building
- You should have a solid grasp of XML.

Organization of This Book

Inside the Microsoft Build Engine is divided into seven parts:

Part I, "Overview," describes all the fundamentals of creating and extending MSBuild project files. Chapter 1, "MSBuild Quick Start," is a brief chapter to get you started quickly with MSBuild. If you are already familiar with MSBuild then you can skip this chapter; its content

will be covered in more detail within chapters 2 and 3. Chapter 2, "MSBuild Deep Dive, Part 1," discusses such things as static properties, static items, targets, tasks, and msbuild .exe usage. Chapter 3, "MSBuild Deep Dive, Part 2," extends on Chapter 2 with dynamic properties, dynamic items, how properties and items are evaluated, importing external files, extending the build process, property functions, and item functions.

Part II, "Customizing MSBuild," covers the two ways that MSBuild can be extended: custom tasks and custom loggers. Chapter 4, "Custom Tasks," covers all that you need to know to create your own custom MSBuild tasks. Chapter 5, "Custom Loggers," details how to create custom loggers and how to attach them to your build process.

Part III, "Advanced MSBuild Topics," discusses advanced MSBuild concepts. Chapter 6, "Batching and Incremental Builds," covers two very important topics, MSBuild batching and supporting incremental building. Batching is the process of categorizing items and processing them in batches. Incremental building enables MSBuild to detect when a target is up-to-date and can be skipped. Incremental building can drastically reduce build times for most developer builds. Chapter 7, "External Tools," provides some guidelines for integrating external tools into the build process. It also shows how NUnit and FXCop can be integrated in the build process in a reusable fashion.

Part IV, "MSBuild Cookbook," consists of two chapters that are devoted to real-world examples. Chapter 8, "Practical Applications, Part 1," contains several examples, including: setting the assembly version, customizing the build process in build labs, handling errors, and replacing values in configuration files. Chapter 9, "Practical Applications, Part 2," covers more examples, most of which are targeted toward developers who are building Web applications using .NET. It includes Web Deployment Projects, starting and stopping services, zipping output files, compressing Javascript file, and encrypting the web.config file.

Part V, "MSBuild in Visual C++ 2010" discusses how MSBuild powers various features of Visual C++ in light of Visual C++ 2010's switch to MSBuild for its build engine. Chapter 10, "MSBuild in Visual C++ 2010, Part 1" introduces the reader to the new .vcxproj file format for Visual C++ projects and illustrates the Visual C++ build process with a block diagram. Then it continues describing its features such as Build Parallelism, Property Sheets, etc. and how MSBuild enables these features. Of particular interest are the new File Tracker based Incremental Build and movement of Visual C++ Directories settings to a property sheet from the earlier Tools > Option page. Chapter 11, "MSBuild in Visual C++ 2010, Part 1" continues the theme of Chapter 10 by describing more Visual C++ features and the underlying MSBuild implementation. This includes Property Pages, Build Customizations, Platform and Platform Toolsets, project upgrade, etc. It also includes a discussion of all the default tasks, targets and property sheets that are shipped with Visual C++ 2010. Of particular interest is the section on multi-targeting which explains the exciting new feature in Visual C++ 2010 which allows building projects using older toolsets such as Visual C++ 2008 toolset. We describe both how to use this feature as well as how this feature is implemented using

MSBuild. Chapter 12, "Extending Visual C++ 2010" describes how you can extend the build system in various ways by leveraging the underlying MSBuild engine. Discussed in this chapter are authoring Build Events, Custom Build Steps, Custom Build Tool to customize Visual C++ build system in a simple way when the full power of MSBuild extensibility is not needed. This is followed by a discussion of adding a custom target and creating a Build Customization which allows you to use the full set of extensibility features offered by MSBuild. One of the important topics in this chapter deals with adding support for a new Platform or a Platform Toolset. The example of using the popular GCC toolset to build Visual C++ projects is used to drive home the point that extending platforms and platform toolsets is easy and natural in Visual C++ 2010.

Part VI, "Team Foundation Build," introduces Team Foundation Build (Team Build) in Chapter 13, "Team Build Quick Start". In this chapter we discuss the architectural components of Team Foundation Build and walkthrough the installation process and the basics of configuring it. In Chapter 14, "Team Build Deep Dive", we examine the process templates that ship with Team Build as well the Team Build API. Chapter 15, "Workflow Foundation Quick Start", introduces the basics of Workflow Foundation to enable customizing the build process. Chapter 16, "Process Template Customization", then leverages this knowledge and explains how to create customized build processes.

Part VII, "Web Deployment Tool" first introduces the Web Deployment Tool (MSDeploy) in Chapter 17 "Web Deployment Tool, Part 1". In that chapter we discuss what MSDeploy is, and how it can be used. We describe how MSDeploy can be used for "online deployment" in which you deploy your application to the target in real time and we discuss "offline deployments" in which you create a package which gets handed off to someone else for the actual deployment. In Chapter 18 "Web Deployment Tool, Part 2" we introduce the Web Publishing Pipeline (WPP). The WPP is the process which your web application follows to go from build output to being deployed on your remote server. It's all captured in a few MSBuild scripts, so it is very customizable and extensible. In that chapter we cover how you can customize and extend the WPP to suit your needs. Then in Chapter 19 "Web Deploy Practical Applications" we show many different examples of how you can use MSDeploy and WPP to deploy your packages. We cover such things as Publishing using MSBuild, parameterizing packages, deploying with Team Build, and a few others.

For Appendices A, B, and C please go to *http://aka.ms/645240/files*.

System Requirements

The following list contains the minimum hardware and software requirements to run the code samples provided with the book.

- .NET 4.0 Framework
- Visual Studio 2010 Express Edition or greater
- 50 MB of available space on the installation drive

For Team Build chapters:

- Visual Studio 2010 Professional
- Some functionality (such as Code Analysis) requires Visual Studio 2010 Premium or Visual Studio 2010 Ultimate
- Access to a server running Team Foundation Server 2010
- Access to a build machine running Team Foundation Build 2010 (Chapter 13 walks you through installing this)
- A trial Virtual PC with Microsoft Visual Studio 2010 and Team Foundation Server 2010 RTM is available from *http://www.microsoft.com/downloads/en/details.aspx?FamilyID=509c3ba1-4efc-42b5-b6d8-0232b2cbb26e*

Code Samples

Download the sample code files from this book's page online:

http://aka.ms/645240/files

Acknowledgements

The authors are happy to share the following acknowledgments.

Sayed Ibrahim Hashimi

Before I wrote my first book I thought that writing a book involved just a few people, but now having written my third book I realize how many different people it takes to successfully launch a book. Unfortunately with books most of the credit goes to the authors, but the others involved deserve much more credit than they are naturally given. As an author, the most we can do is thank them and mention their names here in the acknowledgements section. When I reflect on the writing of this book there are a lot of names, but there is one that stands out in particular, Dan Moseley. Dan is a part of the MSBuild team. He has gone way above and beyond what I could have ever imagined. I've never seen someone peer review a chapter as good, or as fast, as Dan has. Without Dan's invaluable insight the book would simply not be what it is today. In my whole career I've only encountered a few people who are as passionate about what they do as Dan. I hope that I can be as passionate about building products as he is.

Besides Dan I would like to first thank my co-authors and technical editor. William Bartholomew, who wrote the Team Build chapters, is a wonderful guy to work with. He is recognized as a Team Build expert, and I think his depth of knowledge shows in his work. Pavan Adharapurapu wrote the chapters covering Visual C++. When we first started talking about updating the book to cover MSBuild 4.0 to be honest I was a bit nervous. I was nervous because I had not written any un-managed code in more than 5 years, and because of that I knew that I could not write the content on Visual C++ and do it justice. Then we found Pavan. Pavan helped build the Visual C++ project system, and he pours his heart into everything that he does. Looking back I am confident that he was the best person to write those chapters and I am thankful that he was willing. Also I'd like to thank Jason Ward, who wrote a chapter on Workflow Foundation. Jason who has a great background in Workflow Foundation as well as Team Build was an excellent candidate to write that chapter. I started with the authors, but the technical editor, Marc Young deserves the same level of recognition. This having been my third book I was familiar with what a technical editor is responsible for doing. Their primary job is essentially to point out the fact that I don't know what I'm talking about, which Marc did very well. But Marc went beyond his responsibilities. Marc was the one who suggested that we organize all the sample code based on the chapters. At first I didn't really think it was a good idea, but he volunteered to reorganize the content and even redo a bunch of screen shots. I really don't think he knew what he was volunteering for! Now that it is over I wonder if he would volunteer again. I can honestly say that Marc was the best technical editor that I've ever worked with. His attention to detail is incredible, to the point that he was reverse engineering the code to validate some statements that I was making (and some were wrong). Before this book I knew what a technical editor was supposed to be, and now I know what a technical editor can be. Thanks to all of you guys!

As I mentioned at the beginning of this acknowledgement there are many others who came together to help complete this book besides those of us writing it. I'd like to thank Microsoft Press and everyone there who worked on it. I know there were some that were involved that I didn't even know of. I'd like to thank those that I do know of by name. Devon Musgrave, who also worked with us on the first edition, is a great guy to work with. This book really started with him. We were having dinner one night a while back and he said to me something along the lines of "what do you think of updating the book?" I knew that it would be a wonderful project and it was. Iram Nawaz who was the Project Editor of the book was just fantastic. She made sure that we stayed on schedule (sorry for the times I was late ☺) and was a great person to work with. The book wouldn't have made it on time if it was not for her. Along with these guys from Microsoft Press I would like to than the editors; Susan McClung and Nicole Schlutt for their perseverance to correct my bad writing.

There are several people who work on either the MSBuild/MSDeploy/Visual Studio product groups that I would like to thank as well. When the guys who built the technologies you are writing about help you, it brings the book to a whole new level. I would like to thank the following people for giving their valued assistance (in no particular order, and sorry if

I missed anyone); Jay Shrestha, Chris Mann, Andrew Arnott, Vishal Joshi, Bilal Aslam, Faith Allington, Ming Chen, Joe Davis and Owais Shaikh.

William Bartholomew

Firstly I'd like to thank my co-authors, Sayed, Pavan, and Jason, because without their contributions this book would not be as broad as it is. From Microsoft Press I'd like to thank Devon Musgrave, Ben Ryan, Iram Nawaz, Susan McClung, and the art team, for their efforts in converting our ideas into a publishable book. Thanks must go to Marc Young for his technical review efforts in ensuring that the procedures are easily followed, the samples work, and the book makes sense. Finally, I'd like to thank the Team Build Team, in particular Aaron Hallberg and Buck Hodges, for the tireless support.

Pavan Adharapurapu

A large number of people helped make this book happen. I would like to start off by thanking Dan Moseley, my manager at Microsoft who encouraged me to write the book and for providing thorough and detailed feedback for the chapters that I wrote. Brian Tyler, the architect of my team provided encouragement and great feedback. Many people from the Visual C and the project system teams here at Microsoft helped make the book a better one by providing feedback on their areas of expertise. In alphabetical order they are: Olga Arkhipova, Andrew Arnott, Ilya Biryukov, Felix Huang, Cliff Hudson, Renin John, Sara Joiner, Marian Luparu, Chris Mann, Bogdan Mihalcea, Kieran Mockford, Amit Mohindra, Li Shao. Any mistakes that remain are mine.

I would like to thank Devon Musgrave, Iram Nawaz, Susan McClung and Marc Young from Microsoft Press for their guidance and patience.

Finally, I would like to thank my wonderful wife Deepti who provided great support and understanding throughout the many weekends I spent locked up writing and revising the book. Deepti, I promise to make it up to you.

Jason Ward

First of all, I'd like to thank William Bartholomew for giving me the opportunity to contribute to this book. William displays an amazing amount of talent, passion and integrity in all his work. I'm honored to have his friendship as well as the opportunity to work with him on a daily basis.

I'd also like to thank Avi Pilosof and Rich Lowry for giving me the wonderful opportunity to work at Microsoft. From the moment I met them it was clear that moving my family half way around the world was the right thing to do. Their mentorship, passion, friendship

and overarching goal of 'doing the right thing' has only further reinforced that working at Microsoft was everything I had hoped it would be. They are the embodiment of all things good at Microsoft.

Finally I'd like to thank the thousands of people working at Microsoft for producing the wonderful applications and experiences that millions of people around the world use and enjoy on a daily basis. It is truly an honor to work with you as we change the world.

Errata and Book Support

We've made every effort to ensure the accuracy of this book and its companion content. If you do find an error, please report it on our Microsoft Press site:

1. Go to *www.microsoftpressstore.com*.
2. In the Search box, enter the book's ISBN or title.
3. Select your book from the search results.
4. On your book's catalog page, find the Errata & Updates tab

You'll find additional information and services for your book on its catalog page. If you need additional support, please e-mail Microsoft Press Book Support at *mspinput@microsoft.com*.

Please note that product support for Microsoft software is not offered through the addresses above.

We Want to Hear from You

At Microsoft Press, your satisfaction is our top priority, and your feedback our most valuable asset. Please tell us what you think of this book at:

http://www.microsoft.com/learning/booksurvey

The survey is short, and we read *every one* of your comments and ideas. Thanks in advance for your input!

Stay in Touch

Let's keep the conversation going! We're on Twitter: *http://twitter.com/MicrosoftPress*

Part I

Overview

In this part:

Chapter 1

MSBuild Quick Start

When you are learning a new subject, it's exciting to just dive right in and get your hands dirty. The purpose of this chapter is to enable you to do just that. I'll describe all the key elements you need to know to get started using MSBuild. If you're already familiar with MSBuild, feel free to skip this chapter—all of the material presented here will be covered in later areas in the book as well, with the exception of the msbuild.exe usage details.

The topics covered in this chapter include the structure of an MSBuild file, properties, targets, items, and invoking MSBuild. Let's get started.

Project File Details

An MSBuild file—typically called an "MSBuild project file"—is just an XML file. These XML files are described by two XML Schema Definition (XSD) documents that are created by Microsoft: Microsoft.Build.Commontypes.xsd and Microsoft.Build.Core.xsd. These files are located in the %WINDIR%\Microsoft.NET\Framework*vNNNN*\MSBuild folder, where *vNNNN* is the version folder for the Microsoft .NET Framework 2.0, 3.5, or 4.0. If you have a 64-bit machine, then you will find those files in the Framework64 folder as well. (In this book, I'll assume you are using .NET Framework 4.0 unless otherwise specified. As a side note, a new version of MSBuild was not shipped with .NET Framework 3.0.) Microsoft .Build.Commontypes.xsd describes the elements commonly found in Microsoft Visual Studio-generated project files, and Microsoft.Build.Core.xsd describes all the fixed elements in an MSBuild project file. The simplest MSBuild file would contain the following:

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
</Project>
```

This XML fragment will identify that this is an MSBuild file. All your content will be placed inside the Project element. Specifically, we will be declaring *properties*, *items*, *targets*, and a few other things directly under the Project element. When building software applications, you will always need to know two pieces of information: what is being built and what build parameters are being used. Typically, files are being built, and these would be contained in MSBuild items. Build parameters, like Configuration or OutputPath, are contained in MSBuild properties. We'll now discuss how to declare properties as well as targets, and following that we'll discuss items.

Properties and Targets

MSBuild properties are simply key-value pairs. The key for the property is the name that you will use to refer to the property. The value is its value. When you declare static properties, they are always contained in a *PropertyGroup* element, which occurs directly within the *Project* element. We will discuss dynamic properties (those declared and generated dynamically inside targets) in the next chapter. The following snippet is a simple example of declaring static properties:

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <AppServer>\\sayedApp</AppServer>
    <WebServer>\\sayedWeb</WebServer>
  </PropertyGroup>
</Project>
```

As previously stated, the *PropertyGroup* element, inside the *Project* element, will contain all of our properties. The name of a property is the XML tag name of the element, and the value of the property is the value inside the element. In this example, we have declared two properties, AppServer and WebServer, with the values \\sayedApp and \\sayedWeb, respectively. You can create as many *PropertyGroup* elements under the Project tag as you want. The previous fragment could have been defined like this:

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <AppServer>\\sayedApp</AppServer>
  </PropertyGroup>
  <PropertyGroup>
    <WebServer>\\sayedWeb</WebServer>
  </PropertyGroup>
</Project>
```

The MSBuild engine will process all elements sequentially within each *PropertyGroup* in the same manner. If you take a look at a project created by Visual Studio, you'll notice that many properties are declared. These properties have values that will be used throughout the build process for that project. Here is a region from a sample project that I created:

```
<Project DefaultTargets="Build"
    xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <Configuration Condition=" '$(Configuration)' == '' ">Debug</Configuration>
    <Platform Condition=" '$(Platform)' == '' ">AnyCPU</Platform>
    <ProductVersion>8.0.50727</ProductVersion>
    <SchemaVersion>2.0</SchemaVersion>
    <ProjectGuid>{A71540FD-9949-4AC4-9927-A66B84F97769}</ProjectGuid>
    <OutputType>WinExe</OutputType>
    <AppDesignerFolder>Properties</AppDesignerFolder>
    <RootNamespace>WindowsApplication1</RootNamespace>
    <AssemblyName>WindowsApplication1</AssemblyName>
  </PropertyGroup>
```

```
  <PropertyGroup Condition=" '$(Configuration)|$(Platform)' == 'Debug|AnyCPU' ">
    <DebugSymbols>true</DebugSymbols>
    <DebugType>full</DebugType>
    <Optimize>false</Optimize>
    <OutputPath>bin\Debug\</OutputPath>
    <DefineConstants>DEBUG;TRACE</DefineConstants>
    <ErrorReport>prompt</ErrorReport>
    <WarningLevel>4</WarningLevel>
  </PropertyGroup>
 ....
</Project>
```

You can see that values for the output type, the name of the assembly, and many others are defined in properties. Defining properties is great, but we also need to be able to utilize them, which is performed inside targets. We will move on to discuss Target declarations.

MSBuild fundamentally has two execution elements: tasks and targets. A task is the smallest unit of work in an MSBuild file, and a target is a sequential set of tasks. A task must always be contained within a target. Here's a sample that shows you the simplest MSBuild file that contains a target:

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <Target Name="HelloWorld">
  </Target>
</Project>
```

In this sample, we have created a new target named HelloWorld, but it doesn't perform any work at this point because it is empty. When MSBuild is installed, you are given many tasks out of the box, such as Copy, Move, Exec, ResGen, and Csc. You can find a list of these tasks at the MSBuild Task Reference (*http://msdn2.microsoft.com/en-us/library/7z253716.aspx*). We will now use the Message task. This task is used to send a message to the logger(s) that are listening to the build process. In many cases this means a message is sent to the console executing the build. When you invoke a task in an MSBuild file, you can pass its input parameters by inserting XML attributes with values. These attributes will vary from task to task depending on what inputs the task is able to accept. From the documentation of the Message task (*http://msdn2.microsoft.com/en-us/library/6yy0yx8d.aspx*) you can see that it accepts a string parameter named Text. The following snippet shows you how to use the Message task to send the classic message "Hello world!"

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <Target Name="HelloWorld">
    <Message Text="Hello world!" />
  </Target>
</Project>
```

Now we will verify that this works as expected. To do this, place the previous snippet into a file named HelloWorld.proj. Now open a Visual Studio command prompt, found in the Visual Studio Tools folder in the Start menu for Visual Studio. When you open this prompt,

the path to msbuild.exe is already on the path. The command you will be invoking to start MSBuild is msbuild.exe. The basic usage for the command is as follows:

```
msbuild [INPUT_FILE] /t:[TARGETS_TO_EXECUTE]
```

So the command in our case would be

```
msbuild HelloWorld.proj /t:HelloWorld
```

This command says to execute the HelloWorld target, which is contained in the HelloWorld .proj file. The result of this invocation is shown in Figure 1-1.

```
C:\InsideMSBuild\Ch01>msbuild HelloWorld.proj /nologo
Build started 9/24/2010 5:55:31 PM.
Project "C:\InsideMSBuild\Ch01\HelloWorld.proj" on node 1 (default targets).
HelloWorld:
  Hello world!
Done Building Project "C:\InsideMSBuild\Ch01\HelloWorld.proj" (default targets).

Build succeeded.
    0 Warning(s)
    0 Error(s)
```

FIGURE 1-1 Result of HelloWorld target

Note In this example, as well as all others in the book, we specify the /nologo switch. This simply avoids printing the MSBuild version information to the console and saves space in the book. Feel free to use it or not as you see fit.

We can see that the HelloWorld target is executed and that the message "Hello world!" is displayed on the console. The Message task also accepts another parameter, Importance. The possible values for this parameter are high, normal, or low. The Importance value may affect how the loggers interpret the purpose of the message. If you want the message logged no matter the verbosity, use the *high* importance level. We're discussing properties, so let's take a look at how we can specify the text using a property. I've extended the HelloWorld.proj file to include a few new items. The contents are shown here:

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <Target Name="HelloWorld">
    <Message Text="Hello world!" />
  </Target>

  <PropertyGroup>
    <HelloMessage>Hello from property</HelloMessage>
  </PropertyGroup>
  <Target Name="HelloProperty">
    <Message Text="$(HelloMessage)" />
  </Target>
</Project>
```

I have added a new property, HelloMessage, with the value "Hello from property", as well as a new target, HelloProperty. The HelloProperty target passes the value of the property using

the $(*PropertyName*) syntax. This is the syntax you use to evaluate a property. We can see this in action by executing the command `msbuild HelloWorld.proj /t:HelloProperty`. The result is shown in Figure 1-2.

```
C:\InsideMSBuild\Ch01>msbuild HelloWorld.proj /t:HelloProperty /nologo
Build started 9/24/2010 5:59:26 PM.
Project "C:\InsideMSBuild\Ch01\HelloWorld.proj" on node 1 (HelloProperty target(s)).
HelloProperty:
  Hello from property
Done Building Project "C:\InsideMSBuild\Ch01\HelloWorld.proj" (HelloProperty target(s)).

Build succeeded.
    0 Warning(s)
    0 Error(s)
```

FIGURE 1-2 Result of HelloProperty target

As you can see, the value of the property was successfully passed to the Message task. Now that we have discussed targets and basic property usage, let's move on to discuss how we can declare properties whose values are derived from other properties.

To see how to declare a property by using the value of an existing property, take a look at the project file, NestedProperties.proj:

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <Configuration Condition=" '$(Configuration)' == '' ">Debug</Configuration>
    <Platform Condition=" '$(Platform)' == '' ">AnyCPU</Platform>
    <DropLocation>
      \\sayedData\MSBuildExamples\Drops\$(Configuration)\$(Platform)\
    </DropLocation>
  </PropertyGroup>
  <Target Name="PrepareFilesForDrop">
    <Message Text="DropLocation : $(DropLocation)" />
  </Target>
</Project>
```

We can see here that three properties have been declared. On both the Configuration and Platform properties, a *Condition* attribute appears. We'll discuss this attribute later in this chapter. The remaining property, DropLocation, is defined using the values of the two previously declared items. The DropLocation property has three components: a constant value and two values that are derived from the Configuration and Platform properties. When the MSBuild engine sees the $(*PropertyName*) notation, it will replace that with the value of the specified property. So the evaluated value for DropLocation would be \\sayedData\MSBuildExamples\Drops\Debug\AnyCPU\. You can verify that by executing the PrepareFilesForDrop target with msbuild.exe. The reference for properties can be found at *http://msdn.microsoft.com/en-us/library/ms171458.aspx*.

When you use MSBuild, a handful of properties are available to you out of the box that cannot be modified. These are known as reserved properties. Table 1-1 contains all the reserved properties.

TABLE 1-1 Reserved Properties

Name	Description
MSBuildExtensionsPath	The full path where MSBuild extensions are located. By default, this is stored under %programfiles%\msbuild.
MSBuildExtensionsPath32	The full path where MSBuild 32-bit extensions are located. This typically is located under the Program Files folder. For 32-bit machines, this value will be the same as MSBuildExtensionsPath.
MSBuildExtensionsPath64*	The full path where MSBuild 64-bit extensions are located. This typically is under the Program Files folder. For 32-bit machines, this value will be empty.
MSBuildLastTaskResult*	This value holds the return value from the previous task. It will be *true* if the task completed successfully, and *false* otherwise.
MSBuildNodeCount	The number of nodes (processes) that are being used to build the projects. If the /m switch is not used, then this value will be 1.
MSBuildProgramFiles32*	This points to the 32-bit Program Files folder.
MSBuildProjectDefaultTargets	Contains the list of the default targets.
MSBuildProjectDirectory	The full path to the directory where the project file is located.
MSBuildProjectDirectoryNoRoot	The full path to the directory where the project file is located, excluding the root directory.
MSBuildProjectExtension	The extension of the project file, including the period.
MSBuildProjectFile	The name of the project file, including the extension.
MSBuildProjectFullPath	The full path to the project file.
MSBuildProjectName	The name of the project file, without the extension.
MSBuildStartupDirectory	The full path to the folder where the MSBuild process is invoked.
MSBuildThisFile*	The name of the file, including the extension but excluding the path, which contains the target that is currently executing.
MSBuildThisFileDirectory*	This is the full path to the directory that contains the file that is currently being executed.
MSBuildThisFileDirectoryNoRoot*	The same as MSBuildThisFileDirectory, except with the root removed.
MSBuildThisFileExtension*	The extension of the file that is currently executing.
MSBuildThisFileFullPath*	The full path to the file that is currently executing.
MSBuildThisFileName*	The name of the file, excluding the extension and path, of the currently executing file.
MSBuildToolsPath (MSBuildBinPath)	The full path to the location where the MSBuild binaries are located. For MSBuild 2.0, this property is named MSBuildBinPath; in MSBuild 3.5, it is deprecated.
MSBuildToolsVersion	The version of the tools being used to build the project. Possible values include 2.0, 3.5, and 4.0. The default value for this is 2.0.

* Denotes parameters new with MSBuild 4.0.

You would use these properties just as you would properties that you have declared in your own project file. To see an example of this, look at any Visual Studio–generated project file. When you create a new C# project, you will find the import statement `<Import Project="$(MSBuildToolsPath)\Microsoft.CSharp.targets" />` located near the bottom. This import statement uses the MSBuildToolsPath reserved property to resolve the full path to the Microsoft.CSharp.targets file and insert its content at this location. This is the file that drives the build process for C# projects. We will discuss its content throughout the remainder of this book. In Chapter 3, "MSBuild Deep Dive, Part 2," we discuss specifically how the Import statement is processed.

Items

Building applications usually means dealing with many files. Because of this, you use a specific construct when referencing files in MSBuild: items. Items are usually file-based references, but they can be used for other purposes as well. If you create a project using Visual Studio, you may notice that you see many *ItemGroup* elements as well as *PropertyGroup* elements. The *ItemGroup* element contains all the statically defined items. Static item definitions are those declared as a direct child of the *Project* element. Dynamic items, which we discuss in the next chapter, are those defined inside a target. When you define a property, you are declaring a key-value pair, which is a one-to-one relationship. When you declare items, one item can contain a list of many values. In terms of code, a property is analogous to a variable and an item to an array. Take a look at how an item is declared in the following snippet taken from the ItemsSimple.proj file:

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <ItemGroup>
    <SolutionFile Include="..\InsideMSBuild.sln" />
  </ItemGroup>
  <Target Name="PrintSolutionInfo">
    <Message Text="SolutionFile: @(SolutionFile)" />
  </Target>
</Project>
```

In this file, there is an *ItemGroup* that has a subelement, *SolutionFile*. *ItemGroup* is the element type that all statically declared items must be placed within. The name of the subelement, *SolutionFile* in this case, is actually the item type of the item that is created. The *SolutionFile* element has an attribute, Include. This determines what values the item contains. Relating it back to an array, *SolutionFile* is the name of the variable that references the array, and the *Include* attribute is used to populate the array's values. The *Include* attribute can contain the following types of values (or any combination thereof): one distinct value, a list of values delimited with semicolons, or a value using wildcards. In this sample, the *Include* attribute contains one value. When you need to evaluate the contents of an item, you would use the @(*ItemType*) syntax. This is similar to the $(*PropertyName*) syntax for properties. To see this in action, take a look at the PrintSolutionInfo target. This target

passes the value of the item into the Message task to be printed to the console. You can see the result of executing this target in Figure 1-3.

```
C:\InsideMSBuild\Ch01>msbuild ItemsSimple.proj /t:PrintSolutionInfo /nologo
Build started 9/24/2010 6:04:18 PM.
Project "C:\InsideMSBuild\Ch01\ItemsSimple.proj" on node 1 (PrintSolutionInfo target(s)).
PrintSolutionInfo:
  SolutionFile: ..\InsideMSBuild.sln
Done Building Project "C:\InsideMSBuild\Ch01\ItemsSimple.proj" (PrintSolutionInfo target(s)).


Build succeeded.
    0 Warning(s)
    0 Error(s)
```

FIGURE 1-3 PrintSolutionInfo result

In this case, the item *SolutionFile* contains a single value, so it doesn't seem very different from a property because the single value was simply passed to the Message task. Let's take a look at an item with more than one value. This is an extended version of the ItemsSimple .proj file shown earlier:

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <ItemGroup>
    <SolutionFile Include="..\InsideMSBuild.sln" />
  </ItemGroup>
  <Target Name="PrintSolutionInfo">
    <Message Text="SolutionFile: @(SolutionFile)" />
  </Target>

  <ItemGroup>
    <Compile
     Include="Form1.cs;Form1.Designer.cs;Program.cs;Properties\AssemblyInfo.cs" />
  </ItemGroup>
  <Target Name="PrintCompileInfo">
    <Message Text="Compile: @(Compile)" />
  </Target>
</Project>
```

In the modified version, I have created a new item, Compile, which includes four values that are separated by semicolons. The PrintCompileInfo target passes these values to the Message task. When you invoke the PrintCompileInfo target on the MSBuild file just shown, the result will be `Compile: Form1.cs;Form1.Designer.cs;Program.cs;Properties \AssemblyInfo.cs`. It may look like the Message task simply took the value in the Include attribute and passed it to the Message task, but this is not the case. The Message task has a single input parameter, Text, as discussed earlier. This parameter is a string property. Because an item is a multivalued object, it cannot be passed directly into the Text property. It first has to be converted into a string. MSBuild does this for you by separating each value with a semicolon. In Chapter 2, I will discuss how you can customize this conversion process.

An item definition doesn't have to be defined entirely by a single element. It can span multiple elements. For example, the Compile item shown earlier could have been declared like this:

```
<ItemGroup>
    <Compile Include="Form1.cs" />
```

```
    <Compile Include="Form1.Designer.cs" />
    <Compile Include="Program.cs" />
    <Compile Include="Properties\AssemblyInfo.cs" />
  </ItemGroup>
```

In this version, each file is placed into the Compile item individually. These Compile elements could also have been contained in their own *ItemGroup* as well, as shown in the next snippet.

```
<ItemGroup>
  <Compile Include="Form1.cs" />
</ItemGroup>
<ItemGroup>
  <Compile Include="Form1.Designer.cs" />
</ItemGroup>
<ItemGroup>
  <Compile Include="Program.cs" />
</ItemGroup>
<ItemGroup>
  <Compile Include="Properties\AssemblyInfo.cs" />
</ItemGroup>
```

The end result of these declarations would all be the same. You should note that an item is an ordered list, so the order in which values are added to the item is preserved and may in some context affect behavior based on usage. When a property declaration appears after a previous one, the previous value is overwritten. Items act differently from this in that the value of the item is simply appended to instead of being overwritten. We've now discussed two of the three ways to create items. Let's look at using wildcards to create items.

Many times, items refer to existing files. If this is the case, you can use wildcards to automatically include files that meet the constraints of the wildcards. You can use three wildcard elements with MSBuild: ?, *, and **. The ? descriptor is used to denote that exactly one character can take its place. For example, the include declaration of `b?t.cs` could include values such as bat.cs, bot.cs, bet.cs, b1t.cs, and so on. The * descriptor can be replaced with zero or more characters (not including slashes), so the declaration `b*t.cs` could include values such as bat.cs, bot.cs, best.cs, bt.cs, etc. The ** descriptor tells MSBuild to search directories recursively for the pattern. In effect, "*" matches any characters except for "/" while "**" matches any characters, including "/". For example, `Include="src\**\*.cs"` would include all files under the src folder (including subfolders) with the .cs extension.

Item Metadata

Another difference between properties and items is that items can have metadata associated with them. When you create an item, each of its elements is a full-fledged .NET object, which can have a set of values (metadata) associated with it. The metadata that is available on every item, which is called *well-known metadata,* is summarized in Table 1-2.

TABLE 1-2 Well-Known Metadata

Name	Description
Identity	The value that was specified in the Include attribute of the item after it was evaluated.
FullPath	Full path of the file.
RootDir	The root directory to which the file belongs, such as C:\.
Filename	The name of the file, not including the extension.
Extension	The extension of the file, including the period.
RelativeDir	Contains the path specified in the *Include* attribute, up to the final backslash (\).
Directory	Directory of the item, without the root directory.
RecursiveDir	This is the expanded directory path starting from the first ** of the include declaration. If no ** is present, then this value is empty. If multiple ** are present, then RecursiveDir will be the expanded value starting from the first **. This may sound peculiar, but it is what makes recursive copying possible.
ModifiedTime	The last time the file was modified.
CreatedTime	The time the file was created.
AccessedTime	The last time the file was accessed.

To access metadata values, you have to use this syntax:

```
@(ItemType->'%(MetadataName)')
```

ItemType is the name of the item, and MetadataName is the name of the metadata that you are accessing. This is the most basic syntax. To examine what types of values the well-known metadata returns, take a look at the file, WellKnownMetadata.proj, shown here:

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003"
         ToolsVersion="4.0">
  <ItemGroup>
    <src Include="src\one.txt" />
  </ItemGroup>
  <Target Name="PrintWellKnownMetadata">

    <Message Text="===== Well known metadata ====="/>
    <!-- %40 = @ -->
    <!-- %25 = % -->
    <Message Text="%40(src->'%25(FullPath)'): @(src->'%(FullPath)')"/>
    <Message Text="%40(src->'%25(RootDir)'): @(src->'%(RootDir)')"/>
    <Message Text="%40(src->'%25(Filename)'): @(src->'%(Filename)')"/>
    <Message Text="%40(src->'%25(Extension)'): @(src->'%(Extension)')"/>
    <Message Text="%40(src->'%25(RelativeDir)'): @(src->'%(RelativeDir)')"/>
    <Message Text="%40(src->'%25(Directory)'): @(src->'%(Directory)')"/>
    <Message Text="%40(src->'%25(RecursiveDir)'): @(src->'%(RecursiveDir)')"/>
    <Message Text="%40(src->'%25(Identity)'): @(src->'%(Identity)')"/>
    <Message Text="%40(src->'%25(ModifiedTime)'): @(src->'%(ModifiedTime)')"/>
    <Message Text="%40(src->'%25(CreatedTime)'): @(src->'%(CreatedTime)')"/>
    <Message Text="%40(src->'%25(AccessedTime)'): @(src->'%(AccessedTime)')"/>

  </Target>
</Project>
```

Note In order to use reserved characters, such as the % and @, you have to escape them. This is accomplished by the syntax %HV, where *HV* is the hex value of the character. This is demonstrated here with %25 and %40.

Note In this example, we have specified the ToolsVersion value to be 4.0. This determines which version of the MSBuild tools will be used. Although not needed for this sample, we will be specifying this version number from this point forward. The default value is 2.0.

This MSBuild file prints the values for the well-known metadata for the src item. The result of executing the PrintWellKnownMetadata target is shown in Figure 1-4.

```
C:\InsideMSBuild\Ch01>msbuild WellKnownMetadata.proj /t:PrintWellKnownMetadata /nologo
Build started 9/24/2010 6:10:01 PM.
Project "C:\InsideMSBuild\Ch01\WellKnownMetadata.proj" on node 1 (PrintWellKnownMetadata target(s)
).
PrintWellKnownMetadata:
  ===== Well known metadata =====
  @(src->'%(FullPath)'): C:\InsideMSBuild\Ch01\src\one.txt
  @(src->'%(Rootdir)'): C:\
  @(src->'%(Filename)'): one
  @(src->'%(Extension)'): .txt
  @(src->'%(RelativeDir)'): src\
  @(src->'%(Directory)'): InsideMSBuild\Ch01\src\
  @(src->'%(RecursiveDir)'):
  @(src->'%(Identity)'): src\one.txt
  @(src->'%(ModifiedTime)'): 2010-09-08 22:15:12.4218750
  @(src->'%(CreatedTime)'): 2010-09-08 22:15:12.4218750
  @(src->'%(AccessedTime)'): 2010-09-08 22:15:12.4218750
Done Building Project "C:\InsideMSBuild\Ch01\WellKnownMetadata.proj" (PrintWellKnownMetadata targe
t(s)).

Build succeeded.
    0 Warning(s)
    0 Error(s)
```

FIGURE 1-4 PrintWellKnownMetadata result

The figure gives you a better understanding of the well-known metadata's usage. Keep in mind that this demonstrates the usage of metadata in the case where the item contains only a single value.

To see how things change when an item contains more than one value, let's examine MetadataExample01.proj:

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003"
         ToolsVersion="4.0">
  <ItemGroup>
    <Compile Include="*.cs" />
  </ItemGroup>

  <Target Name="PrintCompileInfo">
    <Message Text="Compile fullpath: @(Compile->'%(FullPath)')" />
  </Target>
</Project>
```

In this project file we simply evaluate the FullPath metadata on the Compile item. From the examples with this text, the directory containing this example contains four files: Class1.cs, Class2.cs, Class3.c, and Class4.cs. These are the files that will be contained in the Compile item. Take a look at the result of the PrintCompileInfo target in Figure 1-5.

```
C:\InsideMSBuild\Ch01>msbuild MetadataExample01.proj /t:PrintCompileInfo /nologo
Build started 9/24/2010 6:18:39 PM.
Project "C:\InsideMSBuild\Ch01\MetadataExample01.proj" on node 1 (PrintCompileInfo target(s)).
PrintCompileInfo:
  Compile fullpath: C:\InsideMSBuild\Ch01\Class1.cs;C:\InsideMSBuild\Ch01\Class2.cs;C:\InsideMSBui
  ld\Ch01\Class3.cs;C:\InsideMSBuild\Ch01\Class4.cs
Done Building Project "C:\InsideMSBuild\Ch01\MetadataExample01.proj" (PrintCompileInfo target(s)).


Build succeeded.
    0 Warning(s)
    0 Error(s)
```

FIGURE 1-5 PrintCompileInfo result

You have to look carefully at this output to decipher the result. What is happening here is that a single string is created by combining the full path of each file, separated by a semicolon. The `@(ItemType->'...%()...')` syntax is an "Item Transformation." We will cover transformations in greater detail in Chapter 2. In the next section, we'll discuss conditions. Before we do that, take a minute to look at the project file for a simple Windows application that was generated by Visual Studio. You should recognize many things.

```
<Project DefaultTargets="Build"
xmlns="http://schemas.microsoft.com/developer/msbuild/2003" ToolsVersion="4.0">
  <PropertyGroup>
    <Configuration Condition=" '$(Configuration)' == '' ">Debug</Configuration>
    <Platform Condition=" '$(Platform)' == '' ">AnyCPU</Platform>
    <ProductVersion>8.0.50727</ProductVersion>
    <SchemaVersion>2.0</SchemaVersion>
    <ProjectGuid>{0F34CE5D-2AB0-49A9-8254-B21D1D2EFFA1}</ProjectGuid>
    <OutputType>WinExe</OutputType>
    <AppDesignerFolder>Properties</AppDesignerFolder>
    <RootNamespace>WindowsApplication1</RootNamespace>
    <AssemblyName>WindowsApplication1</AssemblyName>
  </PropertyGroup>
  <PropertyGroup Condition=" '$(Configuration)|$(Platform)' == 'Debug|AnyCPU' ">
    <DebugSymbols>true</DebugSymbols>
    <DebugType>full</DebugType>
    <Optimize>false</Optimize>
    <OutputPath>bin\Debug\</OutputPath>
    <DefineConstants>DEBUG;TRACE</DefineConstants>
    <ErrorReport>prompt</ErrorReport>
    <WarningLevel>4</WarningLevel>
  </PropertyGroup>
  <PropertyGroup Condition=" '$(Configuration)|$(Platform)' == 'Release|AnyCPU' ">
    <DebugType>pdbonly</DebugType>
    <Optimize>true</Optimize>
    <OutputPath>bin\Release\</OutputPath>
    <DefineConstants>TRACE</DefineConstants>
    <ErrorReport>prompt</ErrorReport>
    <WarningLevel>4</WarningLevel>
  </PropertyGroup>
  <ItemGroup>
    <Reference Include="System" />
    <Reference Include="System.Data" />
    <Reference Include="System.Deployment" />
    <Reference Include="System.Drawing" />
    <Reference Include="System.Windows.Forms" />
    <Reference Include="System.Xml" />
  </ItemGroup>
```

```
  <ItemGroup>
    <Compile Include="Form1.cs">
      <SubType>Form</SubType>
    </Compile>
    <Compile Include="Form1.Designer.cs">
      <DependentUpon>Form1.cs</DependentUpon>
    </Compile>
    <Compile Include="Program.cs" />
    <Compile Include="Properties\AssemblyInfo.cs" />
    <EmbeddedResource Include="Properties\Resources.resx">
      <Generator>ResXFileCodeGenerator</Generator>
      <LastGenOutput>Resources.Designer.cs</LastGenOutput>
      <SubType>Designer</SubType>
    </EmbeddedResource>
    <Compile Include="Properties\Resources.Designer.cs">
      <AutoGen>True</AutoGen>
      <DependentUpon>Resources.resx</DependentUpon>
    </Compile>
    <None Include="Properties\Settings.settings">
      <Generator>SettingsSingleFileGenerator</Generator>
      <LastGenOutput>Settings.Designer.cs</LastGenOutput>
    </None>
    <Compile Include="Properties\Settings.Designer.cs">
      <AutoGen>True</AutoGen>
      <DependentUpon>Settings.settings</DependentUpon>
      <DesignTimeSharedInput>True</DesignTimeSharedInput>
    </Compile>
  </ItemGroup>
  <Import Project="$(MSBuildToolsPath)\Microsoft.CSharp.targets" />
  <!-- To modify your build process, add your task
inside one of the targets below and uncomment it.
       Other similar extension points exist,
see Microsoft.Common.targets.
  <Target Name="BeforeBuild">
  </Target>
  <Target Name="AfterBuild">
  </Target>
  -->
</Project>
```

Simple Conditions

When you are building, you often have to make decisions based on conditions. MSBuild allows almost every XML element to contain a conditional statement within it. The statement would be declared in the *Condition* attribute. If this attribute evaluates to *false,* then the element and all its child elements are ignored. In the sample Visual Studio project that was shown at the end of the previous section, you will find the statement `<Configuration Condition=" '$(Configuration)' == '' ">Debug</Configuration>`. In this declaration, the condition is checking to see if the property is empty. If so, then it will be defined; otherwise, the statement will be skipped. This is a method to provide a default overridable value for a property. Table 1-3 describes a few common types of conditional operators.

TABLE 1-3 **Simple Conditional Operators**

Symbol	Description
==	Checks for equality; returns *true* if both have the same value.
!=	Checks for inequality; returns *true* if both do not have the same value.
Exists	Checks for the existence of a file. Returns *true* if the provided file exists.
!Exists	Checks for the nonexistence of a file. Returns *true* if the file provided is not found.

Because you can add a conditional attribute to any MSBuild element (excluding the Otherwise element), this means that we can decide to include entries in items as necessary. For example, when building ASP.NET applications, in some scenarios, you might want to include files that will assist debugging. Take a look at the MSBuild file, ConditionExample01.proj:

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003"
         ToolsVersion="4.0">
  <PropertyGroup>
    <Configuration>Release</Configuration>
  </PropertyGroup>
  <ItemGroup>
    <Content Include="script.js"/>
    <Content Include="script.debug.js" Condition="$(Configuration)=='Debug'" />
  </ItemGroup>

  <Target Name="PrintContent">
    <Message Text="Configuration: $(Configuration)" />
    <Message Text="Content: @(Content)" />
  </Target>
</Project>
```

If we execute the command `msbuild ConditionExample01.proj /t:PrintContent`, the result would be what is shown in Figure 1-6.

```
C:\InsideMSBuild\Ch01>msbuild ConditionExample01.proj /t:PrintContent /nologo
Build started 9/24/2010 6:24:55 PM.
Project "C:\InsideMSBuild\Ch01\ConditionExample01.proj" on node 1 (PrintContent target(s)).
PrintContent:
  Configuration: Release
  Content: script.js
Done Building Project "C:\InsideMSBuild\Ch01\ConditionExample01.proj" (PrintContent target(s)).


Build succeeded.
    0 Warning(s)
    0 Error(s)
```

FIGURE 1-6 PrintContent target result

As you can see, because the Configuration value was not set to Debug, the script.debug.js file was not included in the Content item. Now we will examine the usage of the *Exists* function. To do this, take a look at the target _CheckForCompileOutputs, taken from the Microsoft .Common.targets file, a file included with MSBuild that contains most of the rules for building VB and C# projects:

```
<Target
    Name="_CheckForCompileOutputs">
```

```
    <!--Record the main compile outputs.-->
    <ItemGroup>
        <FileWrites
          Include="@(IntermediateAssembly)"
          Condition="Exists('@(IntermediateAssembly)')" />
    </ItemGroup>

    <!-- Record the .xml if one was produced. -->
    <PropertyGroup>
        <_DocumentationFileProduced
          Condition="!Exists('@(DocFileItem)')">false</_DocumentationFileProduced>
    </PropertyGroup>

    <ItemGroup>
        <FileWrites
          Include="@(DocFileItem)"
          Condition="'$(_DocumentationFileProduced)'=='true'" />
    </ItemGroup>

    <!-- Record the .pdb if one was produced. -->
    <PropertyGroup>
        <_DebugSymbolsProduced
           Condition="!Exists('@(_DebugSymbolsIntermediatePath)')">false
        </_DebugSymbolsProduced>
    </PropertyGroup>

    <ItemGroup>
        <FileWrites
          Include="@(_DebugSymbolsIntermediatePath)"
          Condition="'$(_DebugSymbolsProduced)'=='true'" />
    </ItemGroup>
</Target>
```

From the first FileWrites item definition, the condition is defined as Exists (@(IntermediateAssembly)). This will determine whether the file referenced by the IntermediateAssembly item exists on disk. If it doesn't, then the declaration task is skipped. This was a brief overview of conditional statements, but it should be enough to get you started. Let's move on to learn a bit more about targets.

Default/Initial Targets

When you create an MSBuild file, you will typically create it such that a target, or a set of targets, will be executed most of the time. In this scenario, these targets can be specified as default targets. These targets will be executed if a target is not specifically chosen to be executed. Without the declaration of a default target, the first defined target in the logical project file, after all imports have been resolved, is treated as the default target. A logical project file is one with all Import statements processed. Using default target(s) is how Visual

Studio builds your managed project. If you take a look at Visual Studio–generated project files, you will notice that the Build target is specified as the default target:

```
<Project DefaultTargets="Build"
xmlns="http://schemas.microsoft.com/developer/msbuild/2003" ToolsVersion="4.0">
...
</Project>
```

As mentioned previously, you can have either one target or many targets be your default target(s). If the declaration contains more than one, the target names need to be separated by a semicolon. When you use a command such as `msbuild ProjectFile.proj`, because you have not specified a target to execute, the default target(s) will be executed. It's important to note that the list of DefaultTargets will be preserved, not modified, through an Import, provided that a project previously processed hasn't had a DefaultTargets list. This is one difference between DefaultTargets and InitialTargets. Values for InitialTargets are aggregated for all imports because each file may have its own initialization checks.

These targets listed in InitialTargets will always be executed even if the project file is imported by other project files. Similar to default targets, the initial targets list is declared as an attribute on the *Project* element with the name InitialTargets. If you take a look at the Microsoft.Common.targets file, you will notice that the target _CheckForInvalidConfigurationAndPlatform is declared as the initial target. This target will perform a couple sanity checks before allowing the build to continue. I would strongly encourage the use of default targets. InitialTargets should be used to verify initial conditions before the build starts and raises an error or warning if applicable. Next, we will discuss the command-line usage of the msbuild.exe command.

MSBuild.exe Command-Line Usage

In this section, we'll discuss the most important options when invoking msbuild.exe. When you invoke the msbuild.exe executable, you can pass many parameters to customize the process. We'll first take a look at the options that are available with MSBuild 2.0, and then we'll discuss what differences exist for MSBuild 3.5 and MSBuild 4.0. Table 1-4 summarizes the parameters you can pass to msbuild.exe. Many commands include a short version that can be used; these versions are listed in the table within parentheses.

TABLE 1-4 MSBuild.exe Command-Line Switches

Switch	Description
/help (/?)	Displays the usage information for msbuild.exe.
/nologo	Suppresses the copyright and startup banner.
/version (/ver)	Displays version information.
@file	Used to pick up response file(s) for parameters.

Switch	Description
/noautoresponse (/noautoresp)	Used to suppress automatically, including msbuild.rsp as a response file.
/target (/t)	Used to specify which target(s) should be built. If specifying more than one target, they should each be separated by a semicolon. Commas are valid separators, but semicolons are the ones most commonly used.
/property:<n>=<v> (/p)	Used to specify properties. If providing more than one property, they should each be separated by a semicolon. Property values should be specified in the format: *name=value*. These values would supersede any static property definitions. Commas are valid separators, but semicolons are the ones most commonly used.
/verbosity (/v)	Sets the verbosity of the build. The options are quiet (q), minimal (m), normal (n), detailed (d), and diagnostic (diag). This is passed to each logger, and the logger is able to make its own decision about how to interpret it.
/validate (/val)	Used to ensure that the project file is in the correct format before the build is started.
/logger (/l)	Attaches the specified logger to the build. This switch can be provided multiple times to attach any number of loggers. Also, you can pass parameters to the loggers with this switch.
/consoleloggerparameters (/clp)	Used to pass parameters to the console logger.
/noconsolelogger (/noconlog)	Used to suppress the usage of the console logger, which is otherwise always attached.
/filelogger (/fl)	Attaches a file logger to the build.
/fileloggerparameters (/flp)	Passes parameters to the file logger. If you want to attach multiple file loggers, you do so by specifying additional parameters in the switches /flp1, /flp2, /flp3, and so on.
/distributedFileLogger (/dl)	Used to attach a distributed logger. This is an advanced switch that you will most likely not use and that could have been excluded altogether.
/maxcpucount (/m)	Sets the maximum number of processes that should be used by msbuild.exe to build the project.
/ignoreprojectextensions (/ignore)	Instructs MSBuild to ignore the extensions passed.
/toolsversion (/tv)	Specifies the version of the .NET Framework tools that should be used to build the project.
/nodeReuse (/nr)	Used to specify whether nodes should be reused or not. Typically, there should be no need to specify this; the default value is optimal.

Switch	Description
/preprocess (/pp)*	This will output the complete logical file to either the console or to a specified file. To have the result written out to the file, use the syntax `/pp:file`. Usually, this file will build just as if you were building the original project (there are exceptions though, such as $(MSBuildThisFile)). The real purpose of this is to help diagnose a problem with the build by avoiding the need to jump between many different files. For example, if a particular property is getting overwritten somewhere, it is much easier to search for it in the single "preprocessed" file than it is to search for it in the many imported files.
/detailedSummary (/ds)*	It displays information about how the projects were scheduled to different CPUs. You can use this to help figure out how to make the build faster. For example, you can use this to determine which project was stalling other projects.

* Denotes parameters new with MSBuild 4.0.

From Table 1-4, the most commonly used parameters are target, property, and logger. You might also be interested in using the FileLogger switch. To give you an example, I will use an MSBuild file that we discussed earlier, the ConditionExample01.proj file. Take a look at the following command that will attach the file logger to the build process: `msbuild ConditionExample01.proj /fl`. Because we didn't specify the name of the log file to be written to, the default, msbuild.log, will be used. Using this same project file, let's see how to override the Configuration value. From that file, the Configuration value would be set to Release, but we can override it from the command line with the following statement: `msbuild ConditionExample01.proj /p:Configuration=Debug /t:PrintContent`. In this command, we are using the */p* (property) switch to provide a property value to the build engine, and we are specifying to execute the PrintContent target. The result is shown in Figure 1-7.

```
C:\InsideMSBuild\Ch01>msbuild ConditionExample01.proj /p:Configuration=Debug /t:PrintContent /nolog
o
Build started 9/24/2010 6:42:28 PM.
Project "C:\InsideMSBuild\Ch01\ConditionExample01.proj" on node 1 (PrintContent target(s)).
PrintContent:
  Configuration: Debug
  Content: script.js;script.debug.js
Done Building Project "C:\InsideMSBuild\Ch01\ConditionExample01.proj" (PrintContent target(s)).


Build succeeded.
    0 Warning(s)
    0 Error(s)
```

FIGURE 1-7 Specifying a property from the command line

The messages on the console show that the value for Configuration was indeed Debug, and as expected, the debug JavaScript file was included in the Content item. Now that you know the basic usage of the msbuild.exe command, we'll move on to the last topic: extending the build process.

Extending the Build Process

With versions of Visual Studio prior to 2005, the build was mostly a black box. The process by which Visual Studio built your applications was internal to the Visual Studio product itself. The only way you could customize the process was to use execute commands for pre- and post-build events. With this, you were able to embed a series of commands to be executed. You were not able to change how Visual Studio built your applications. With the advent of MSBuild, Visual Studio has externalized the build process and you now have complete control over it. Since MSBuild is delivered with the .NET Framework, Visual Studio is not required to build applications. Because of this, we can create build servers that do not need to have Visual Studio installed. We'll examine this by showing how to augment the build process. Throughout the rest of this book, we will describe how to extend the build process in more detail.

The pre- and post-build events mentioned earlier are still available, but you now have other options. The three main ways to add a pre- or post-build action are:

- Pre- and post-build events
- Override BeforeBuild/AfterBuild target
- Extend the BuildDependsOn list

The pre- and post-build events are the same as described previously. This is a good approach for backward compatibility and ease of use. Configuring this using Visual Studio doesn't require knowledge of MSBuild. Figure 1-8 shows the Build Events tab on the ProjectProperties page.

Here, you can see the two locations for the pre- and post-build events toward the center of the image. The dialog that is displayed is the post-build event command editor. This helps you construct the command. You define the command here, and MSBuild executes it for you at the appropriate time using the Exec task (*http://msdn2.microsoft.com/en-us/library/x8zx72cd.aspx*). Typically, these events are used to copy or move files around before or after the build.

Using the pre- and post-build event works fairly well if you want to execute a set of commands. If you need more control over what is occurring, you will want to manually modify the project file itself. When you create a new project using Visual Studio, the project file generated is an MSBuild file, which is an XML file. You can use any editor you choose, but if you use Visual Studio, you will have IntelliSense when you are editing it! With your solution loaded in Visual Studio, you can right-click the project, select Unload Project, right-click the project again, and select Edit. If you take a look at the project file, you will notice this statement toward the bottom of the file.

```
<!-- To modify your build process, add your task inside one
       of the targets below and uncomment it.
       Other similar extension points exist, see Microsoft.Common.targets.
  <Target Name="BeforeBuild">
  </Target>
  <Target Name="AfterBuild">
  </Target>
  -->
```

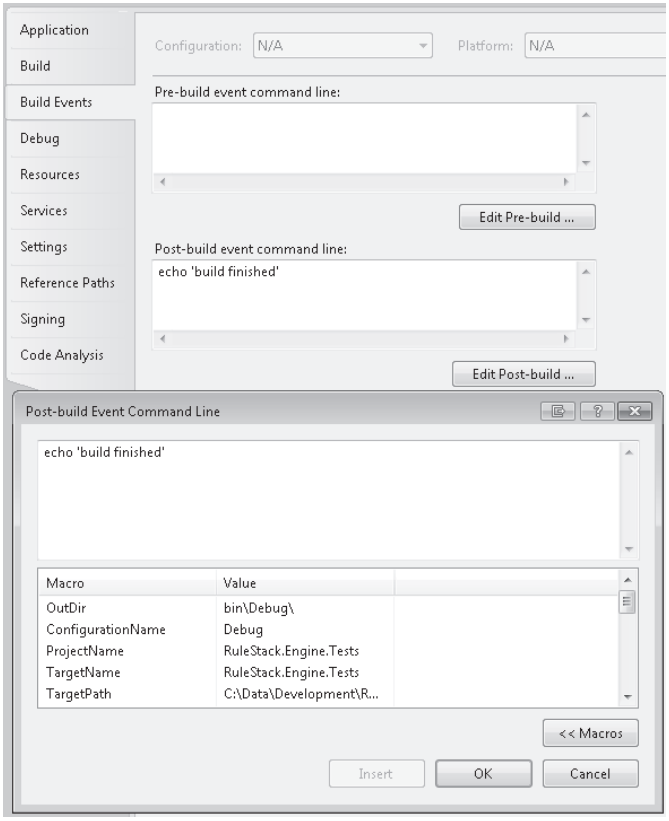

FIGURE 1-8 Build Events tab

From the previous snippet, we can see that there are predefined targets designed to handle these types of customizations. We can simply follow the directions from the project file, by defining the BeforeBuild or AfterBuild target. You will want to make sure that these definitions are **after** the *Import* element for the Microsoft.*.targets file, where * represents the language of the project you are editing. For example, you could insert the following AfterBuild target:

```
<Target Name="AfterBuild">
  <Message Text="Build has completed!" />
</Target>
```

When the build has finished, this target will be executed and the message '`Build has completed`!' will be passed to the loggers. We will cover the third option, extending the BuildDependsOn list, in Chapter 3.

In this chapter, we have covered many features of MSBuild, including properties, items, targets, and tasks. Now you should have all that you need to get started customizing your build process. From this point on, the remainder of the book will work on filling in the details that were left out here so that you can become an MSBuild expert!

Chapter 2

MSBuild Deep Dive, Part 1

In the previous chapter, we gave a brief overview of all the key elements in MSBuild. In this chapter and the next, we'll examine most of those ideas in more detail. We'll discuss properties, items, targets, tasks, transformations, and much more. After you have completed this chapter, you will have a solid grasp of how to create and modify MSBuild files to suit your needs. After the next chapter, we'll explore ways to extend MSBuild as well as some advanced topics.

What is MSBuild? MSBuild is a general-purpose build system created by Microsoft and is used to build most Microsoft Visual Studio projects. MSBuild is shipped with the Microsoft .NET Framework. What this means is that you do *not* need to have Visual Studio installed in order to build your applications. This is very beneficial because you don't need to purchase licenses of Visual Studio for dedicated build machines, and it makes configuring build machines easier. Another benefit is that MSBuild will be installed on many machines. If .NET Framework 2.0 or later is available on a machine, so is a version of MSBuild. The following terms have been used to identify an MSBuild file: MSBuild file, MSBuild project file, MSBuild targets file, MSBuild script, etc. When you create an MSBuild file, you should follow these conventions for specifying the extension of the file:

- **.proj** A project file
- **.targets** A file that contains shared targets, which are imported into other files
- **.props** Default settings for a build process
- **.tasks** A file that contains UsingTask declarations

An MSBuild file is just an XML file. You can use any editor you choose to create and edit MSBuild files. The preferred editor is Visual Studio, because it provides IntelliSense on the MSBuild files as you are editing them. This IntelliSense will greatly decrease the amount of time required to write an MSBuild file. The IntelliSense is driven by a few XML Schema Definition (XSD) files. These XSD files, which are all in Visual Studio's XML directory, are Microsoft.Build.xsd, Microsoft.Build.Core.xsd, and Microsoft.Build.Commontypes.xsd. The Microsoft.Build.xsd file imports the other two files, and provides an extension point for task developers to include their own files. The Microsoft.Build.Core.xsd file describes all the fundamental elements that an MSBuild file can contain.

Microsoft.Build.Commonttypes.xsd defines all known elements; this is mainly used to describe the elements that Visual Studio–generated project files can contain. The XSD that is used is not 100 percent complete, but in most cases you will not notice that. Now that we have discussed what it takes to edit an MSBuild file, let's discuss properties in detail. If you are not familiar with invoking msbuild.exe from the command line, take a look back at Chapter 1, "MSBuild Quick Start"; this is not covered again here.

Properties

MSBuild has two main constructs for representing data: properties and items. A property is a key-value pair. Each property can have exactly one value. An item list differs from a property in that it can have many values. In programming terms, a property is similar to a scalar variable, and an item list is similar to an array variable, whose order is preserved. Properties are declared inside the *Project* element in a *PropertyGroup* element. We'll now take a look at how properties are declared. The following file, Properties01.proj, demonstrates declaration and usage of a property.

```
<Project xmlns=http://schemas.microsoft.com/developer/msbuild/2003
         ToolsVersion="4.0">

  <PropertyGroup>
    <Configuration>Debug</Configuration>
  </PropertyGroup>

  <Target Name="PrintConfig">
    <Message Text="Config: $(Configuration)" />
  </Target>

</Project>
```

As stated previously, we needed a *PropertyGroup* element, and the Configuration property was defined inside of that. By doing this we have created a new property named Configuration and given it the value *Debug*. When you create properties, you are not limited to defining only one property per *PropertyGroup* element. You can define any number of properties inside a single *PropertyGroup* element. In the target PrintConfig, the Message task is invoked in order to print the value of the Configuration property. If you are not familiar with what a target is, refer back to Chapter 1, "MSBuild Quick Start." You can execute that target with the command `msbuild.exe Properties01.proj /t:PrintConfig`. The results of this command are shown in Figure 2-1.

```
C:\InsideMSBuild\Ch02>msbuild Properties01.proj /t:PrintConfig /nologo
Build started 9/28/2010 9:52:48 PM.
Project "C:\InsideMSBuild\Ch02\Properties01.proj" on node 1 (PrintConfig target(s)).
PrintConfig:
  Config: Debug
Done Building Project "C:\InsideMSBuild\Ch02\Properties01.proj" (PrintConfig target(s)).

Build succeeded.
    0 Warning(s)
    0 Error(s)
```

FIGURE 2-1 PrintConfig target results

From the result in Figure 2-1, we can see that the correct value for the Configuration property was printed as expected. As properties are declared, their values are recorded in a top-to-bottom order. What this means is that if a property is defined, and then defined again, the last value will be the one that is applied. Take a look at a modified version of the previous example; this one is contained in the Properties02.proj file.

```
<Project xmlns=http://schemas.microsoft.com/developer/msbuild/2003
         ToolsVersion="4.0">

  <PropertyGroup>
    <Configuration>Debug</Configuration>
  </PropertyGroup>

  <PropertyGroup>
    <Configuration>Release</Configuration>
  </PropertyGroup>

  <Target Name="PrintConfig">
    <Message Text="Config: $(Configuration)" />
  </Target>

</Project>
```

In this example, we have declared the Configuration property once again, after the existing declaration, and specified that it have the value *Release*. Because the new value is declared *after* the previous one, we would expect the new value to hold. If you execute the PrintConfig target on this file, you will see that this is indeed the case. Properties in MSBuild can be declared any number of times. This is not an erroneous condition, and there is no way to detect this. Now we will look at another version of the previous file, a slightly modified one. Take a look at the contents of the following Properties03.proj file.

```
<Project xmlns=http://schemas.microsoft.com/developer/msbuild/2003
         ToolsVersion="4.0">

  <PropertyGroup>
    <Configuration>Debug</Configuration>
  </PropertyGroup>

  <PropertyGroup>
    <Configuration>Release</Configuration>
  </PropertyGroup>

  <Target Name="PrintConfig">
    <Message Text="Config: $(Configuration)"/>
  </Target>

  <PropertyGroup>
    <Configuration>CustomRelease</Configuration>
  </PropertyGroup>

</Project>
```

This example is a little different in the sense that there is a value for Configuration declared after the PrintConfig target. That value is *CustomRelease*. So if we execute the PrintConfig target, what should be the result, *Release* or *CustomRelease*? We can execute `msbuild.exe Properties03.proj /t:PrintConfig` to find out. The results of this command are shown in Figure 2-2.

```
C:\InsideMSBuild\Ch02>msbuild Properties03.proj /t:PrintConfig /nologo
Build started 9/28/2010 9:54:10 PM.
Project "C:\InsideMSBuild\Ch02\Properties03.proj" on node 1 (PrintConfig target(s)).
PrintConfig:
  Config: CustomRelease
Done Building Project "C:\InsideMSBuild\Ch02\Properties03.proj" (PrintConfig target(s)).


Build succeeded.
    0 Warning(s)
    0 Error(s)
```

FIGURE 2-2 PrintConfig result for Properties03.proj

As can be seen from the results in Figure 2-2, the value for Configuration that was printed was *CustomRelease*! How is this possible? It was defined after the PrintConfig target! This is because MSBuild processes the entire file for properties and items ***before*** any targets are executed. You can imagine all the properties being in a dictionary, and as the project file is processed, its values are placed in the dictionary. Property names are *not* case sensitive, so Configuration and CoNfiguratION would refer to the same property. After the entire file, including imported files, is processed, all the final values for statically declared properties and items have been resolved. Once all the properties and items have been resolved, targets are allowed to execute. We'll take a closer look at this process in the section entitled "Property and Item Evaluation," in Chapter 3, "MSBuild Deep Dive, Part 2."

Note We will discuss importing files in Chapter 3.

Environment Variables

We have described the basic usage of properties. Now we'll discuss a few other related topics. When you are building your applications, sometimes you might need to extract values from environment variables. This is a lot simpler than you might imagine if you use MSBuild. You can access values, just as you would properties, for environment variables. For example, take a look at the following project file, Properties04.proj.

```
<Project xmlns=http://schemas.microsoft.com/developer/msbuild/2003
         ToolsVersion="4.0">

  <Target Name="PrintEnvVar">
    <Message Text="Temp: $(Temp)" />
    <Message Text="Windir: $(windir)" />
    <Message Text="VS100COMNTOOLS: $(VS100COMNTOOLS)" />
  </Target>

</Project>
```

In this example, we can see that no properties have been declared and no other files are imported. Inside the target, PrintEnvVar, we can see that we have made a few messages to print the values of some properties. These values are being pulled from the environment variables. When you use the $(*PropertyName*) syntax to retrieve a value, MSBuild will first look to see if

there is a corresponding property. If there is, its value is returned. If there isn't, then it will look at the environment variables for a variable with the provided name. If such a variable exists, its value is returned. If you execute the command `msbuild.exe Properties04.proj /t:PrintEnvVar` you should see a result similar to that shown in Figure 2-3.

```
C:\InsideMSBuild\Ch02>msbuild Properties04.proj /t:PrintEnvVar /nologo
Build started 9/28/2010 9:57:37 PM.
Project "C:\InsideMSBuild\Ch02\Properties04.proj" on node 1 (PrintEnvVar target(s)).
PrintEnvVar:
  Temp: C:\Users\Ibrahim\AppData\Local\Temp
  Windir: C:\WINDOWS
  VS100COMNTOOLS: C:\Program Files\Microsoft Visual Studio 10.0\Common7\Tools\
Done Building Project "C:\InsideMSBuild\Ch02\Properties04.proj" (PrintEnvVar target(s)).

Build succeeded.
    0 Warning(s)
    0 Error(s)
```

FIGURE 2-3 Environment variable usage

As demonstrated in Figure 2-3, the values for the appropriate environment variables were printed as expected.

> **Note** When MSBuild starts (that is, when msbuild.exe starts or when Visual Studio starts), all the environment variables and their values are captured *at that time*. So if a value for an environment variable changes after that, it will not be reflected in the build. Also, you should be aware that each project is isolated from environment variable changes and changes to the current directory that are made by other projects.

If you don't have Visual Studio 2010 installed on the machine running this file, then the value may be empty for the VS100COMNTOOLS property. As we just saw, you can get the value for an environment variable by using the property notation. Assigning a value to a property that has the same name as an environment variable has no effect on the environment variable itself. The $(*PropertyName*) notation can get a value from an environment variable, but it will never assign values to environment variables. Let's move on to discuss reserved properties.

Reserved Properties

There are a fixed number of reserved properties. These are properties that are globally available to every MSBuild script and that can never be overwritten. These properties are provided to users by the MSBuild engine itself, and many of them are very useful. These are summarized in Table 2-1.

TABLE 2-1 Reserved Properties

Name	Description
MSBuildProjectDirectory	The full path to the directory where the project file is located.
MSBuildProjectDirectoryNoRoot	The full path to the directory where the project file is located, excluding the root (for example, c:\).

Name	Description
MSBuildProjectFile	The name of the project file, including the extension.
MSBuildProjectExtension	The extension of the project file, including the period.
MSBuildProjectFullPath	The full path to the project file.
MSBuildProjectName	The name of the project file, without the extension.
MSBuildProjectDefaultTargets	Contains a list of the default targets.
MSBuildExtensionsPath	The full path to where MSBuild extensions are located. This is typically under the Program Files folder. Note that now this always points to the 32-bit location.
MSBuildExtensionsPath32	The full path to where MSBuild 32 bit extensions are located. This is typically under the Program Files folder. For 32-bit machines, this value will be the same as MSBuildExtensionsPath.
MSBuildExtensionsPath64 *	The full path to where MSBuild 64-bit extensions are located. This is typically under the Program Files folder. For 32-bit machines, this value will be empty.
MSBuildNodeCount	The maximum number of nodes (processes) that are being used to build the project. If the /m switch is not used, then this value will be 1. If you use the /m switch without specifying a number of nodes, then the default is the number of CPUs available.
MSBuildStartupDirectory	The full path to the folder where the MSBuild process was invoked.
MSBuildToolsPath (MSBuildBinPath)	The full path to the location where the MSBuild binaries are located. In MSBuild 2.0, this property is named MSBuildBinPath and is deprecated in MSBuild 3.5 and later. MSBuildBinPath and MSBuildToolsPath have the same value, but you should use only MSBuildToolsPath.
MSBuildToolsVersion	The version of the tools being used to build the project. Possible values include 2.0, 3.5, and 4.0. The default value is 2.0.
MSBuildLastTaskResult *	This contains *true* if the last executed task was a success (*task returned true*) and *false* if it ended in a failure. If a task fails, typically the build stops unless you specified `ContinueOnError="true"`.
MSBuildProgramFiles32 *	This contains the path to the 32-bit Program Files folder. To get the value for the default Program Files folder, use `$(ProgramFiles)`.
MSBuildThisFile *	Contains the file name, including the extension, of the file that contains the property usage. This differs from MSBuildProjectFile in that MSBuildProjectFile always refers to the file that was invoked, not any imported file name.
MSBuildThisFileDirectory *	The path of the folder of the file that uses the property. This is useful if you need to define any items whose location you know relative to the targets file.

Name	Description
MSBuildThisFileDirectoryNoRoot *	Same as MSBuildThisFileDirectory without the root (for example, InsideMSBuild\Ch02 instead of C:\InsideMSBuild\Ch02).
MSBuildThisFileExtension *	The extension of the file referenced by MSBuildThisFile.
MSBuildThisFileFullPath *	The full path to the file that contains the usage of the property.
MSBuildThisFileName *	The name of the file, excluding the extension, to the file that contains usage of the property.
MSBuildOverrideTasksPath *	MSBuild 4.0 introduces override tasks, which are tasks that force themselves to be used instead of any other defined task with the same name, and this property points to a file that contains the overrides. The override tasks feature is used internally to help MSBuild 4.0 work well with other versions of MSBuild.

* denotes parameters new with MSBuild 4.0.

Note You are allowed to override the values for MSBuildExtensionsPath, as well as the 32- and 64-bit variants. This is useful in case you check shared tasks into source control and want to use those files.

You would use these properties in the same way as you would any other properties. In order to understand what types of values these properties are set to, I have created the following sample file, ReservedProperties01.proj, to print out all these values.

```
<Project xmlns=http://schemas.microsoft.com/developer/msbuild/2003
         ToolsVersion="4.0">
  <Target Name="PrintReservedProperties">
    <Message Text="MSBuildProjectDirectory: $(MSBuildProjectDirectory)" />
    <Message Text="MSBuildProjectDirectoryNoRoot: $(MSBuildProjectDirectoryNoRoot)" />
    <Message Text="MSBuildProjectFile: $(MSBuildProjectFile)" />
    <Message Text="MSBuildProjectExtension: $(MSBuildProjectExtension)" />
    <Message Text="MSBuildProjectFullPath: $(MSBuildProjectFullPath)" />
    <Message Text="MSBuildProjectName: $(MSBuildProjectName)" />
    <Message Text="MSBuildToolsPath: $(MSBuildToolsPath)" />
    <Message Text="MSBuildProjectDefaultTargets: $(MSBuildProjectDefaultTargets)" />
    <Message Text="MSBuildExtensionsPath: $(MSBuildExtensionsPath)" />
    <Message Text="MSBuildExtensionsPath32: $(MSBuildExtensionsPath32)" />
    <Message Text="MSBuildExtensionsPath64: $(MSBuildExtensionsPath64)" />
    <Message Text="MSBuildNodeCount: $(MSBuildNodeCount)" />
    <Message Text="MSBuildStartupDirectory: $(MSBuildStartupDirectory)" />
    <Message Text="MSBuildToolsPath: $(MSBuildToolsPath)" />
    <Message Text="MSBuildToolsVersion: $(MSBuildToolsVersion)" />
    <Message Text="MSBuildLastTaskResult: $(MSBuildLastTaskResult)" />
    <Message Text="MSBuildProgramFiles32: $(MSBuildProgramFiles32)" />
    <Message Text="MSBuildThisFile: $(MSBuildThisFile)" />
    <Message Text="MSBuildThisFileDirectory: $(MSBuildThisFileDirectory)" />
    <Message Text="MSBuildThisFileDirectoryNoRoot: $(MSBuildThisFileDirectoryNoRoot)" />
    <Message Text="MSBuildThisFileExtension: $(MSBuildThisFileExtension)" />
    <Message Text="MSBuildThisFileFullPath: $(MSBuildThisFileFullPath)" />
```

```
    <Message Text="MSBuildThisFileName: $(MSBuildThisFileName)" />
    <Message Text="MSBuildOverrideTasksPath: $(MSBuildOverrideTasksPath)" />
  </Target>
</Project>
```

If you execute this build file using the command `msbuild.exe ReservedProperties01.proj /t:PrintReservedProperties`, you would see the results shown in Figure 2-4.

```
C:\InsideMSBuild\Ch02>msbuild ReservedProperties01.proj /t:PrintReservedProperties /nologo
Build started 9/28/2010 10:03:50 PM.
Project "C:\InsideMSBuild\Ch02\ReservedProperties01.proj" on node 1 (PrintReservedProperties targe
t(s)).
PrintReservedProperties:
  MSBuildProjectDirectory: C:\InsideMSBuild\Ch02
  MSBuildProjectDirectoryNoRoot: InsideMSBuild\Ch02
  MSBuildProjectFile: ReservedProperties01.proj
  MSBuildProjectExtension: .proj
  MSBuildProjectFullPath: C:\InsideMSBuild\Ch02\ReservedProperties01.proj
  MSBuildProjectName: ReservedProperties01
  MSBuildToolsPath: C:\WINDOWS\Microsoft.NET\Framework\v4.0.30319
  MSBuildProjectDefaultTargets:
  MSBuildExtensionsPath: C:\Program Files\MSBuild
  MSBuildExtensionsPath32: C:\Program Files\MSBuild
  MSBuildExtensionsPath64:
  MSBuildNodeCount: 1
  MSBuildStartupDirectory: C:\InsideMSBuild\Ch02
  MSBuildToolsPath: C:\WINDOWS\Microsoft.NET\Framework\v4.0.30319
  MSBuildToolsVersion: 4.0
  MSBuildLastTaskResult: true
  MSBuildProgramFiles32: C:\Program Files
  MSBuildThisFile: ReservedProperties01.proj
  MSBuildThisFileDirectory: C:\InsideMSBuild\Ch02\
  MSBuildThisFileDirectoryNoRoot: InsideMSBuild\Ch02\
  MSBuildThisFileExtension: .proj
  MSBuildThisFileFullPath: C:\InsideMSBuild\Ch02\ReservedProperties01.proj
  MSBuildThisFileName: ReservedProperties01
  MSBuildOverrideTasksPath:
Done Building Project "C:\InsideMSBuild\Ch02\ReservedProperties01.proj" (PrintReservedProperties t
arget(s)).

Build succeeded.
    0 Warning(s)
    0 Error(s)
```

FIGURE 2-4 Reserved properties

Most of these values are straightforward. You should note that the values relating to the MSBuild file, with the exception of those starting with *MSBuildThis*, are always qualified relative to the MSBuild file that is invoking the entire process. This becomes clear when you use the *Import* element to import additional MSBuild files. For the *MSBuildThis* properties, those values always refer to the file that contains the element. We will take a look at importing external files in the next chapter.

Command-Line Properties

You can also provide properties through the command line. As stated in Chapter 1, we can use the /property switch (short version /p) to achieve this. We will see how this works now. When you use the /p switch, you must specify the values in the format `/p:<n>=<v>`, where *<n>* is the name of the property and *<v>* is its value. You can provide multiple values by separating the pairs by a semicolon or a comma. We will demonstrate a simple case with the following project file, Properties05.proj.

```
<Project xmlns=http://schemas.microsoft.com/developer/msbuild/2003
         ToolsVersion="4.0">
```

```
  <Target Name="PrintInfo">
    <Message Text="AssemblyName: $(AssemblyName)" />
    <Message Text ="OutputPath: $(OutputPath)" />
  </Target>

</Project>
```

Because there are no values for AssemblyName or OutputPath, it would be pointless to execute this MSBuild file. If we pass them in through the command line, you can see their values. If you specify values for AssemblyName and OutputPath with the command `msbuild.exe Properties05.proj /t:PrintInfo /p:AssemblyName=Sedo.Namhu .Common;OutputPath="deploy\Release\\"`, then the result would be what is shown in Figure 2-5.

```
C:\InsideMSBuild\Fundamentals>msbuild.exe Properties05.proj /t:PrintInfo /p:AssemblyName=Sedo.Namhu
.Common;OutputPath="deploy\Release\"  /nologo
Build started 5/13/2010 11:57:34 PM.
Project "C:\InsideMSBuild\Fundamentals\Properties05.proj" on node 1 (PrintInfo target(s)).
PrintInfo:
  AssemblyName: Sedo.Namhu.Common
  OutputPath: "deploy\Release\"
Done Building Project "C:\InsideMSBuild\Fundamentals\Properties05.proj" (PrintInfo target(s)).

Build succeeded.
    0 Warning(s)
    0 Error(s)
```

FIGURE 2-5 PrintInfo result for Properties05.proj

From Figure 2-5, we can see that the values for the properties that were provided at the command line were successfully passed through. Note in this example that we passed the OutputPath contained in quotes and the end is marked with \\ because \" is an escaped quote mark ("). In this case, the quotes are optional, but if you are passing values containing spaces, then they are required. When you provide a value for a property through the command line, it takes precedence over all other static property declarations. To demonstrate this, take a look at a different version of this file, Properties06.proj, with the values defined.

```
<Project xmlns=http://schemas.microsoft.com/developer/msbuild/2003
         ToolsVersion="4.0">

  <PropertyGroup>
    <AssemblyName>assemblyName</AssemblyName>
  </PropertyGroup>

  <Target Name="PrintInfo">
    <Message Text="AssemblyName: $(AssemblyName)" />
    <Message Text ="OutputPath: $(OutputPath)" />
  </Target>

  <PropertyGroup>
    <OutputPath>outputPath</OutputPath>
  </PropertyGroup>

</Project>
```

In this file, we have specified a value for both AssemblyName and OutputPath. To show that the location of the property with respect to targets doesn't affect the result, I have placed one value at the beginning of the file and the other at the end. If you execute the command `msbuild.exe Properties06.proj /t:PrintInfo /p:AssemblyName=Sedo.Namhu .Common;OutputPath="deploy\Release\\",` the result would be the same as that shown in Figure 2-5. Command-line properties are special properties and have some special behavior that you should be aware of:

- Command-line properties cannot have their values changed (except through dynamic properties, which is covered in the next section).
- The values get passed to all projects through the MSBuild task.
- Their values take precedence over all other property type values, including environment variables and toolset properties. The MSBuild toolset defines what version of the MSBuild tools will be used. For example, you can use v2.0, v3.5, or v4.0.

Thus far, we have covered pretty much everything you need to know about static properties. Now we'll move on to discuss dynamic properties.

Dynamic Properties

When you create properties in your build scripts, static properties will be good enough most of the time. But there are many times when you need to either create new properties or to modify the values of existing properties during the build within targets. These types of properties can be called dynamic properties. Let's take a look at how we can create and use these properties.

In MSBuild 2.0, there was only one way to create dynamic properties, and that was using the CreateProperty task. In MSBuild 3.5 and 4.0, there is a much cleaner approach that you should use, which we cover right after our discussion on the CreateProperty task. Before we discuss how we can use CreateProperty, we have to discuss how to get a value from a task out to the MSBuild file calling it. When a task exposes a value to MSBuild, this is known as an Output property. MSBuild files can extract output values from tasks using the *Output* element. The *Output* element must be placed inside the tags of the task to extract the value. A task can see only those items and properties passed into it explicitly. This is by design and makes it easier to maintain and reuse tasks. To demonstrate this, take a look at the following project file.

```
<Project xmlns=http://schemas.microsoft.com/developer/msbuild/2003
         ToolsVersion="4.0">

  <Target Name="PrintProperty">
    <Message Text="AssemblyName: $(AssemblyName)" />

    <CreateProperty Value="Sedodream.Build.Tasks">
```

```
      <Output TaskParameter="Value" PropertyName="AssemblyName" />
    </CreateProperty>

    <Message Text="AssemblyName: $(AssemblyName)" />
  </Target>

</Project>
```

In this file, the PrintProperty target first prints the value for AssemblyName, which hasn't been defined so it should be empty. Then the CreateProperty task is used to define the AssemblyName property. Let's take a close look at this so we can fully understand the invocations. The statement <CreateProperty Value="Sedodream.Build.Tasks"> invokes CreateProperty and initializes the property named Value to Sedodream.Build.Tasks. The inner statement, <Output TaskParameter="Value" PropertyName="AssemblyName" />, populates the MSBuild property AssemblyName with the value for the .NET property Value. The *Output* element must declare a TaskParameter, which is the name of the task's .NET property to output, and can either contain a value of *PropertyName* or *ItemName,* depending on whether it is supposed to output a property or item, respectively. In this case, we are emitting a property so we use the value *PropertyName*. Looking back at the example shown previously, we would expect that after the CreateProperty task executes, the property AssemblyName will be set to *Sedodream.Build.Tasks*. The result of the `PrintProperty` target is shown in Figure 2-6.

```
C:\InsideMSBuild\Ch02>msbuild Properties07.proj /t:PrintProperty /nologo
Build started 9/28/2010 10:24:24 PM.
Project "C:\InsideMSBuild\Ch02\Properties07.proj" on node 1 (PrintProperty target(s)).
PrintProperty:
  AssemblyName:
  AssemblyName: Sedodream.Build.Tasks
Done Building Project "C:\InsideMSBuild\Ch02\Properties07.proj" (PrintProperty target(s)).

Build succeeded.
    0 Warning(s)
    0 Error(s)
```

FIGURE 2-6 PrintProperty results

From the results shown in Figure 2-6, we can see that the value for AssemblyName was set, as expected, by the CreateProperty task. In this example, we are creating a property that did not exist previously, but the CreateProperty task also can modify the value for existing properties. If you use the task to output a value to a property that already exists, then it will be overwritten. This is true unless a property is reserved. Command-line parameters cannot be overwritten by statically declared properties, only by properties within targets.

If you are using MSBuild 3.5 or 4.0, you can use the CreateProperty task, but there is a cleaner method. You can place *PropertyGroup* declarations directly inside of targets. With this new approach, you can create static and dynamic properties in the same manner. The cleaner version of the previous example is shown as follows. This is contained in the Properties08.proj file.

```
<Project xmlns=http://schemas.microsoft.com/developer/msbuild/2003
         ToolsVersion="4.0">
```

```
  <Target Name="PrintProperty">
    <Message Text="AssemblyName: $(AssemblyName)" />

    <PropertyGroup>
      <AssemblyName>Sedodream.Build.Tasks</AssemblyName>
    </PropertyGroup>

    <Message Text="AssemblyName: $(AssemblyName)" />
  </Target>

</Project>
```

The results of the preceding project file are identical to the example shown in Properties07 .proj, but the syntax is much clearer. This is the preferred approach to creating dynamic properties. This syntax is not supported by MSBuild 2.0, so be sure not to use it in such files. Now that we have thoroughly covered properties, we'll move on to discuss items in detail.

Items

When software is being built, files and directories are used heavily. Because of the usage and importance of files and directories, MSBuild has a specific construct to support these. This construct is items. In the previous section, we covered properties. As stated previously, in programming terms, properties can be considered a regular scalar variable. This is because a property has a unique name and a single value. An item can be thought of as an array. This is because an item has a single name but can have multiple values. Properties use *PropertyGroup* to declare properties; similarly, items use an *ItemGroup* element. Take a look at the following very simple example from Items01.proj.

```
<Project xmlns=http://schemas.microsoft.com/developer/msbuild/2003
         ToolsVersion="4.0">

  <ItemGroup>
    <SourceFiles Include="src\one.txt" />
  </ItemGroup>

  <Target Name="Print">
    <Message Text="SourceFiles: @(SourceFiles)" />
  </Target>

</Project>
```

As stated previously, statically declared items will be inside an *ItemGroup* element. The value for the *Include* attribute determines what values get assigned to the item. Of the few types of values that can be assigned to the *Include* attribute, we'll start with the simplest. The simplest value for *Include* is a text value. In the previous sample, one item, SourceFiles, is declared. The SourceFiles item is set to include one file, which is located at src\one.txt. To get the value of

an item, you use the @(*ItemType*) syntax. In the Print target this is used on the SourceFiles item. The result of the `Print` target is shown in Figure 2-7.

```
C:\InsideMSBuild\Ch02>msbuild Items01.proj /t:Print /nologo
Build started 9/28/2010 10:26:35 PM.
Project "C:\InsideMSBuild\Ch02\Items01.proj" on node 1 (Print target(s)).
Print:
  SourceFiles: src\one.txt
Done Building Project "C:\InsideMSBuild\Ch02\Items01.proj" (Print target(s)).

Build succeeded.
    0 Warning(s)
    0 Error(s)
```

FIGURE 2-7 Print target result for Items01.proj

From the result shown in Figure 2-7, you can see that the file was assigned to the SourceFiles item as expected. From this example, an item seems to behave exactly as a property; this is because we assigned only a single value to the item. The behavior changes when there are more values assigned to the item. The following example is a modified version of the previous example. This modified version is contained in the Items02.proj file.

```
<Project xmlns=http://schemas.microsoft.com/developer/msbuild/2003
         ToolsVersion="4.0">

  <ItemGroup>
    <SourceFiles Include="src\one.txt" />
    <SourceFiles Include="src\two.txt" />
  </ItemGroup>

  <Target Name="Print">
    <Message Text="SourceFiles: @(SourceFiles)" />
  </Target>

</Project>
```

In this version, the SourceFiles item type is declared twice. When more than one item declaration is encountered, the values are appended to each other instead of overwritten like properties. Alternatively, you could have declared the SourceFiles item on a single line by placing both values inside the *Include* attribute, separated by a semicolon. So the previous sample would be equivalent to the following one. With respect to item declarations, ordering is significant and preserved.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003"
         ToolsVersion="4.0">

  <ItemGroup>
    <SourceFiles Include="src\one.txt;src\two.txt" />
  </ItemGroup>

  <Target Name="Print">
    <Message Text="SourceFiles: @(SourceFiles)" />
  </Target>

</Project>
```

If you execute the Print target on this file, the result will be what is shown in Figure 2-8.

```
C:\InsideMSBuild\Ch02>msbuild Items02.proj /t:Print /nologo
Build started 9/28/2010 10:28:25 PM.
Project "C:\InsideMSBuild\Ch02\Items02.proj" on node 1 (Print target(s)).
Print:
  SourceFiles: src\one.txt;src\two.txt
Done Building Project "C:\InsideMSBuild\Ch02\Items02.proj" (Print target(s)).


Build succeeded.
    0 Warning(s)
    0 Error(s)
```

FIGURE 2-8 Print target results for Items02.proj

In this version, we have supplied two values into the SourceFiles item. If you look at the documentation for the Message task, you will notice that the Text property is a string. Fundamentally, there are two types of values in MSBuild: single-valued values and multi-valued values. These are known as *scalar values* and *vector values,* respectively. Properties are scalar values, and items are vector values. What happens when we have a vector value that we need to pass to a task that is accepting only scalar values? MSBuild will first flatten the item before sending it to the task. The value that is passed to the Text property on the Message can be only a single-valued parameter, not a multi-valued one. The *@(ItemType)* operator flattens the SourceFiles item for us, before it is sent into the task. When using *@(ItemType)*, if there is only one value inside the item, that value is used. If there is more than one value contained by the item, then all values are combined, separated by a semicolon by default. Flattening an item is the most basic example of an item transformation. We'll discuss this topic, and using custom separators, in more detail in the section entitled "Item Transformations," later in this chapter. For now, let's move on to see how items are more commonly used.

Note MSBuild doesn't recognize file types by extension as some other build tools do. Also, be aware that item lists do not have to point to files; they can be any type of list-based value. We will see examples of this throughout this book.

Copy Task

A very common scenario for builds is copying a set of files from one place to another. How can we achieve this with MSBuild? There are several ways to do this, which we will demonstrate in this chapter. Before we discuss how to copy the files, we'll first take a close look at the Include statement of an item. I have created some sample files shown in the following tree, which we will use for the remainder of the chapter.

```
C:\InsideMSBuild\Ch02
¦
¦ ...
¦
+---src
```

```
   ¦  one.txt
   ¦  two.txt
   ¦  three.txt
   ¦  four
   ¦
   +---sub
         sub_one.txt
         sub_two.txt
         sub_three.txt
         sub_four.txt
```

Previously, I said that three types of values can be contained in the Include declaration of an item:

1. A single value
2. Multiple values separated by a "**;**"
3. Declared using wildcards

We have shown how 1 and 2 work, so now we'll discuss 3—using wildcards to declare items. These wildcards always resolve values to items on disk. There are three wildcard declarations: *, **, and ?. You may already be familiar with these from usage in other tools, but we will quickly review them once again here. The * descriptor is used to declare that either zero or more characters can be used in its place. The ** descriptor is used to search directories recursively, and the ? is a placeholder for only one character. Effectively, the "*" descriptor matches any characters except for "/" while "**" descriptor matches any characters, including "/". For example, if *file.*proj* used this declaration, the following values would meet the criteria: *file.csproj, file.vbproj, file.vdproj, file.vcproj, file.proj, file.mproj, file.1proj,* etc. In contrast, *file.?proj* will allow only one character to replace the ? character. Therefore, from the previous list of matching names, only *file.mproj* and *file.1proj* meet those criteria. We will examine the ** descriptor shortly in an example. Take a look at the snippet from the following Copy01 .proj file.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003"
         ToolsVersion="4.0">

  <ItemGroup>
    <SourceFiles Include="src\*" />
  </ItemGroup>

  <Target Name="PrintFiles">
    <Message Text="SourceFiles: @(SourceFiles)" />
  </Target>

</Project>
```

In this example, we have used the * syntax to populate the SourceFiles item. Using this syntax, we would expect all the files in the src\ folder to be placed into the item. In order to verify this, you can execute the PrintFiles target. If you were to do this, the result would be