Learn by doing: less theory, more results

.NET 4.0 Generics

Enhance the type safety of your code and create applications easily using Generics in .NET Framework 4.0

Foreword by
Dr. Don Syme, Principal Researcher, Microsoft Research, Cambridge, U.K.
Dr. Andrew Kennedy, Researcher, Microsoft Research, Cambridge, U.K.

Beginner's Guide

Sudipta Mukherjee

[PACKT] PUBLISHING

.NET 4.0 Generics

Beginner's Guide

Enhance the type safety of your code and create applications easily using Generics in the .NET 4.0 Framework

Sudipta Mukherjee

[PACKT]
PUBLISHING

BIRMINGHAM - MUMBAI

.NET 4.0 Generics
Beginner's Guide

Copyright © 2012 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: January 2012

Production Reference: 1190112

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-84969-078-2

www.packtpub.com

Cover Image by Asher Wishkerman (a.wishkerman@mpic.de)

Credits

Author

Sudipta Mukherjee

Reviewers

Atul Gupta

WEI CHUNG, LOW

Antonio Radesca

Acquisition Editor

David Barnes

Lead Technical Editor

Meeta Rajani

Technical Editors

Veronica Fernandes

Copy Editor

Laxmi Subramanian

Project Coordinator

Vishal Bodwani

Proofreader

Joanna McMahon

Indexer

Monica Ajmera Mehta

Rekha Nair

Graphics

Manu Joseph

Production Coordinator

Alwin Roy

Cover Work

Alwin Roy

Foreword

It is my pleasure to write the foreword to a book which will introduce you to the world of generic programming with C# and other .NET languages. You will be able to learn a lot from this book, as it introduces you to the elegant power of generic programming in C#. Through it, you will become a better C# programmer, and a better programmer in all future languages you might choose to use.

It is now almost 10 years since .NET Generics was first described in publications from Microsoft Research, Cambridge, a project I was able to lead and contribute to, and six years since it was released in product form in C# 2.0. In this foreword, I would like to take a moment to review the importance of .NET Generics in the history of programming languages, and the way it continues to inspire a new generation of programmers.

When we began the design of C# and .NET Generics, generic programming was not new. However, it was considered to be outside the mainstream, and attempts to change that with C++ templates and proposals for Java Generics were proving highly problematic for practitioners. At Microsoft Research, we pride ourselves on solving problems at their core. The three defining core features of .NET Generics as we designed them were efficient generics over value types with code generation and sharing managed by the virtual machine, reified run-time types, and language neutrality.

These technical features are now widely acknowledged to represent the "right" fundamental design choices for programming language infrastructure. They are not easy to design or build, and they are not easy to deliver, and when Microsoft Research embarked on this project, we believe we put the .NET platform many years ahead of its rivals. The entire credit goes to Microsoft and people such as Bill Gates, Eric Rudder, and Anders Hejlsberg for taking the plunge to push this into our range of programming languages. However, without the prototyping, research, engineering, and incessant advocacy by Microsoft Research, C# and .NET Generics would never be in their current form.

Let's take some time to examine why this was important. First, .NET Generics represents the moment where strongly typed and functional programming entered the mainstream. .NET Generics enabled C# to become more functional (through LINQ, Lambdas, and generic collections), and it enabled a new class of strongly typed, fully functional .NET languages (such as F#) to thrive. Further, .NET Generics also enabled new key programming techniques, such as Async programming in F# 2.0 and C# 5.0, and Rx programming for reactive systems. Even though you may not realize it, you'll have learned a lot of functional programming by the end of this book.

Next, .NET Generics categorically proved that strongly typed object-oriented programming can integrate seamlessly with generic programming. It is hard to describe the extent to which .NET Generics managed to defeat the "object fundamentalists" of the 1990s (who want a world where there is nothing but classes). These people, many still occupying powerful positions in the software industry, seemed satisfied with a world where programmers are less productive, and programs less efficient, in the name of orthodoxy. Today, no practicing programmer or language designer with experience of .NET Generics would design a strongly typed programming language that does not include Generics. Further, almost every .NET API now features the use of .NET Generics, and it has become an essential weapon in the programmer's toolkit for solving many problems.

Finally, and for me most importantly, .NET Generics represents the victory of pragmatic beauty over pragmatic ugliness. In the eyes of many, alternative solutions to the problem of generic programming such as Java's "erasure" of Generics are simply unpleasant "hacks". This leads to reduced productivity when using those languages. In contrast, .NET Generics is perhaps the most smoothly integrated advanced programming language feature ever constructed. It integrates with reflection, .NET NGEN pre-compilation, debugging, and run-time code generation. I've had many people e-mail me to say that .NET Generics is their favorite programming language feature. That is what language research is all about.

I trust you will learn a great deal from this book, and enjoy the productivity that comes from C#, and .NET languages such as F#.

Dr. Don Syme
Principal Researcher,
Microsoft Research, Cambridge, U.K.

Generic types are more than just lists of `T`. Functional programmers have known this for a long time. C++ programmers who use templates knew this too. But 10 years ago when Don Syme and I first designed and prototyped the Generics feature of the .NET run-time, most mainstream developers were constrained by the rudimentary type systems of languages such as Visual Basic and Java, writing type-generic code only by resorting to casting tricks or worse. In that space, it's hard to conceive of myriad uses of generic types beyond lists and simple collections, and it's fair to say that there was some resistance to our design! Fortunately, some forward thinkers in Microsoft's .NET run-time team regarded Generics in managed languages as more than an academic indulgence, and committed substantial resources to completing a first-class implementation of Generics that is deeply embedded in the run-time languages and tools.

We've come a long way in 10 years! Managed code frameworks make liberal use of generic types, ranging from obvious collection types such as List and Dictionary, through `action' types such as Func and IEnumerable, to more specialized use of Generics such as Lazy initialization. Blogs and online forums are full of discussions on sophisticated topics such as variance and circular constraints. And if it weren't for Generics, it's hard to see how newer language features such as LINQ, or even complete languages such as F#, could have got off the ground.

Coming back, Generics really does start with `List<T>`, and this book sensibly begins from there. It then takes a leisurely tour around the zoo of generic types in the .NET Framework and beyond, to Power Collections and C5. The style is very much one of exploration: the reader is invited to experiment with Generics, prodding and poking a generic type through its methods and properties, and thereby understand the type and solve problems by using it. As someone whose background is in functional programming, in which the initial experience is very much like experimenting with a calculator, I find this very appealing. I hope you like it as much as I do.

Dr. Andrew Kennedy
Researcher,
Microsoft Research, Cambridge, U.K.

About the Author

Sudipta Mukherjee was born in Kolkata and migrated to Bangalore, the IT capital of India, to assume the position of a Senior Research Engineer in a renowned research lab. He is an Electronics Engineer by education and a Computer Engineer/Scientist by profession and passion. He graduated in 2004 with a degree in Electronics and Communication Engineering. He has been working with .NET Generics since they first appeared in the .NET Framework 2.0.

He has a keen interest in data structure, algorithms, text processing, natural language processing ,programming language, tools development, and game development.

His first book on data structure using the C programming language has been well received. Parts of the book can be read on Google Books at `http://goo.gl/pttSh`. The book was also translated into Simplified Chinese available on Amazon at `http://goo.gl/lc536`.

He is an active blogger and an open source enthusiast. He mainly blogs about programming and related concepts at `sudipta.posterous.com`. Inspired by several string processing methods in other languages, he created an open source string processing framework for .NET, available for free at `stringdefs.codeplex.com`.

He lives in Bangalore with his wife. He can be reached via e-mail at `sudipto80@yahoo.com` and via Twitter at `@samthecoder`.

Acknowledgement

Books like this cannot be brought to life by the author alone. I want to take this opportunity to thank all the people who were involved in this book in any way.

First of all, I want to thank Microsoft Research for bringing Generics into the .NET Framework. Great work guys. I have used STL in C++ and Collections in Java. But I can say without being biased that Generics in .NET is the smartest implementation of generic programming paradigm that I have ever come across. Without that, I wouldn't have anything to write about.

I owe a big "Thank You" to the Senior Acquisition Editor and Publisher David Barnes at Packt Publishing for offering me this opportunity to write for them. I want to thank Vishal Bodwani and Meeta Rajani, also from Packt Publishing, for their great support. Everytime I missed a deadline, they helped me get back on track. Thanks for bearing with me. Last but not the least, I want to thank my Technical Editors Snehal and Veronica who painstakingly corrected all the mistakes, did all the formatting, without which the book would not have been possible. Thanks a lot.

I have no words to express my gratitude towards Don and Andrew for taking time off to read the manuscript and their kind words. Thank you Don. Thank you Andrew.

I want to thank all the reviewers of the book. Thanks for all your great feedback. It really made the book better.

My wife, Mou, motivated me to write this book. She stood by me when I needed her throughout all these months. Thank you sweetheart. Last but not the least, I can't thank my mom Dipali and dad Subrata enough for finding the love of my life and always being supportive. Thank you mom. Thank you dad.

About the Reviewers

Atul Gupta, is currently a Principal Technology Architect at Infosys' Microsoft Technology Center. He also has close to 15 years of experience working on Microsoft technologies. His expertise spans user interface technologies, and he currently focuses on Windows Presentation Foundation (WPF) and Silverlight technologies. Other technologies of interest to him are Touch (Windows 7), Deepzoom, Pivot, Surface, and Windows Phone 7.

He recently co-authored the book "*ASP.NET 4 Social Networking*", *Packt Publishing* (`http://www.packtpub.com/asp-net-4-social-networking/book`). His prior interest areas were COM, DCOM, C, VC++, ADO.NET, ASP.NET, AJAX, and ASP.NET MVC.

He has also authored papers for industry publications and websites, some of which are available on Infosys' Technology Showcase (`http://www.infosys.com/microsoft/resource-center/pages/technology-showcase.aspx`). Along with colleagues from Infosys, Atul is also an active blogger (`http://www.infosysblogs.com/microsoft`). Being actively involved in professional Microsoft online communities and developer forums, Atul has received Microsoft's Most Valuable Professional award for multiple years in a row.

WEI CHUNG, LOW, a Technical Lead in BizTalk and .NET, and a MCT, MCPD, MCITP, MCTS, and MCSD.NET, works with ResMed (NYSE: RMD), at its Kuala Lumpur, Malaysia campus. He is also a member of PMI, certified as a PMP. He started working on Microsoft .NET very early on and has been involved in development, consultation, and corporate training in the areas of business intelligence, system integration, and virtualization. He has been working for the Bursa Malaysia (formerly Kuala Lumpur Stock Exchange) and Shell IT International previously, which prepared him with rich integration experience across different platforms.

He strongly believes that great system implementation delivers precious value to the business, and integration of various systems across different platforms shall always be a part of it, just as people from different cultures and diversities are able to live in harmony in most of the major cities.

Antonio Radesca has over 15 years of programming experience. He has a degree in Computer Science and is interested in architectures, programming languages, and enterprise development. He has worked at some of the most important Italian companies, especially at Microsoft .NET Framework as a Developer and an Architect. His expertise spans .NET programming to mobile development on iOS, Android, and Windows Phone.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

Thanks for picking up this book. This is an example-driven book. You will learn about several generic containers and generic algorithms available in the .NET Framework and a couple of other majorly accepted APIs such as Power Collections and C5 by building several applications and programs.

Towards the end, several benchmarkings have been carried out to identify the best container for the job at hand.

What this book covers

Chapter 1, *Why Generics?*, introduces .NET Generics. We will examine the need for the invention of Generics in the .NET Framework. If you start with a feel of "Why should I learn Generics?", you will end with a feeling of "Why didn't I till now?"

Chapter 2, *Lists*, introduces you to several kinds of lists that .NET Generics has to offer. There are simple lists and associative lists. You shall see how simple lists can deliver amazing results avoiding any typecasting woes and boosting performance at the same time.

Chapter 3, *Dictionaries*, explains the need for associative containers and introduces you to the associative containers that .NET has to offer. If you need to keep track of one or multiple dependent variables while one independent variable changes, you need a dictionary. For example, say you want to build a spell check or an autocomplete service, you need a dictionary. This chapter will walk you through this. Along the way, you will pick up some very important concepts.

Chapter 4, *LINQ to Objects*, explains LINQ to objects using extension methods. LINQ or Language Integrated Query is a syntax that allows us to query collections unanimously. In this chapter, we will learn about some standard LINQ Standard Query Operators (LSQO) and then use them in unison to orchestrate an elegant query for any custom need.

Chapter 5, *Observable Collections*, introduces observable collections. Observing events on collections has been inherently difficult. That's going to change forever, thanks to observable collections. You can now monitor your collections for any change; whether some elements are added to the collection, some of them are deleted, change locations, and so on. In this chapter, you will learn about these collections.

Chapter 6, *Concurrent Collections*, covers concurrent collections that appeared in .NET 4.0. Multi-threaded applications are ubiquitous and that's the new expectation of our generation. We are always busy and impatient, trying to get a lot of things done at once. So concurrency is here and it is here to stay for a long time. Historically, there was no inbuilt support for concurrency in generic collections. Programmers had to ensure concurrency through primitive thread locking. You can still do so, but you now have an option to use the concurrent version of generic collections that support concurrency natively. This greatly simplifies the code. In this chapter, you will learn how to use them to build some useful applications such as simulating a survey engine.

Chapter 7, *Power Collections*, introduces several generic algorithms in `PowerCollections` and some handy generic containers. This collection API came from Wintellect (`www.wintellect.com`) at the time when .NET Generics was not big and had some very useful collections. However, now .NET Generics has grown to support all those types and even more. So that makes most of the containers defined in `PowerCollections` outdated. However, there are a lot of good general purpose generic algorithms that you will need but which are missing from the .NET Generics API. That's the reason this chapter is included. In this chapter, you will see how these generic algorithms can be used with any generic container seamlessly.

Chapter 8, *C5 Collections,* introduces the C5 API. If you come from a Java background and are wondering where your hash and tree-based data structures, are this is the chapter to turn to. However, from the usage perspective, all the containers available in C5 can be augmented with generic containers available in the .NET Framework. You are free to use them. This API is also home to several great generic algorithms that make life a lot easier. In this chapter, you will walk through the different collections and algorithms that C5 offers.

Chapter 9, *Patterns, Practices, and Performance*, covers some best practices when dealing with Generics and introduces the benchmarking strategy. In this chapter, we will use benchmarking code to see how different generic containers perform and then declare a winner in that field. For example, benchmarking shows that if you need a set, then `HashSet<T>` in the .NET Framework is the fastest you can get.

Appendix A, *Performance Cheat Sheet*, is a cheat sheet with all the performance measures for all containers. Keeping this handy would be extremely useful when you want to decide which container to use for the job at hand.

Appendix B, *Migration Cheat Sheet*, will show you how to migrate code from STL/JCF/PowerCollections/.NET 1.0 to the latest .NET Framework-compliant code. Migration will never be easier. Using this cheat sheet, it will be a no brainer. This is great for seasoned C++, Java, or .NET developers who are looking for a quick reference to .NET Generics in the latest framework.

What you need for this book

You will need the following software to use this book:

- Visual Studio 2010 (any version will do, I have used the Ultimate Trial version)
- LINQPad

Instructions to download this software are given in the respective chapters where they are introduced.

Who this book is for

This book is for you, if you want to know what .NET Generics is all about and how it can help solve real-world problems. It is assumed that readers are familiar with C# program constructs such as variable declaration, looping, branching, and so on. No prior knowledge in .NET Generics or generic programming is required.

This book also offers handy migration tips from other generic APIs available in other languages, such as STL in C++ or JCF in Java. So if you are trying to migrate your code to the .NET platform from any of these, then this book will be helpful.

Last but not the least, this book ends with generic patterns, best practices, and performance analysis for several generic containers. So, if you are an architect or senior software engineer and have to define coding standards, this will be very handy as a showcase of proofs to your design decisions.

Conventions

In this book, you will find several headings appearing frequently.

To give clear instructions of how to complete a procedure or task, we use:

Time for action – heading

1. Action 1
2. Action 2
3. Action 3

Instructions often need some extra explanation so that they make sense, so they are followed with:

What just happened?

This heading explains the working of tasks or instructions that you have just completed.

You will also find some other learning aids in the book, including:

Pop quiz – heading

These are short, multiple choice questions intended to help you test your own understanding.

Have a go hero – heading

These set practical challenges and give you ideas for experimenting with what you have learned.

You will also find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "Suppose, I want to maintain a list of my students, then we can do that by using `ArrayList` to store a list of such `Student` objects."

A block of code is set as follows:

```
private T[] Sort<T>(T[] inputArray)
{
      //Sort input array in-place
      //and return the sorted array
      return inputArray;
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
Enumerable.Range(1, 100).Reverse().ToList()
  .ForEach(n => nums.AddLast(n));
```

Any command-line input or output is written as follows:

```
Argument 1: cannot convert from 'int[]' to 'float[]'
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Then go to the **File** menu to create a console project."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title through the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/support`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website, or added to any list of existing errata, under the Errata section of that title.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Why Generics?

A cat and a dog shouldn't share a bed. Neither should integers and floats.

Thanks for picking up the book! This means you care for Generics. This is similar to dropping a plastic bag in favor of our lonely planet.

We are living in an interesting era, where more and more applications are data driven. To store these different kinds of data, we need several data structures. Although the actual piece of data is different, that doesn't always necessarily mean that the type of data is different. For example, consider the following situations:

Let's say, we have to write an application to pull in tweets and Facebook wall updates for given user IDs. Although these two result sets will have different features, they can be stored in a similar list of items. The list is a generic list that can be programmed to store items of a given type, at compile time, to ensure type safety. This is also known as parametric polymorphism.

In this introductory chapter, I shall give you a few reasons why Generics is important.

An analogy

Here is an interesting analogy. Assume that there is a model *hand pattern*:

If we fill the pattern with *clay*, we get a *clay-modeled hand*. If we fill it with *bronze*, we get a hand model replica *made of bronze*. Although the material in these two hand models are very different, they share the same pattern (or they were created using the same algorithm, if you would agree to that term, in a broader sense).

Reason 1: Generics can save you a lot of typing

Extrapolating the algorithm part, let's say we have to implement some sorting algorithm; however, data types can vary for the input. To solve this, you can use overloading, as follows:

```
//Overloaded sort methods
private int[] Sort(int[] inputArray)
{
    //Sort input array in-place
    //and return the sorted array
    return inputArray;
}
private float[] Sort(float[] inputArray)
{
    //Sort input array in-place
    //and return the sorted array
    return inputArray;
}
```

However, you have to write the same code for all numeric data types supported by .NET. That's bad. Wouldn't it be cool if the compiler could somehow be instructed at compile time to yield the right version for the given data type at runtime? That's what Generics is about. Instead of writing the same method for all data types, you can create one single method with a symbolic data type. This will instruct the compiler to yield a specific code for the specific data type at runtime, as follows:

```
private T[] Sort<T>(T[] inputArray)
{
      //Sort input array in-place
      //and return the sorted array
      return inputArray;
}
```

`T` is short for **Type**. If you replace `T` with anything, it will still compile; because it's the symbolic name for the generic type that will get replaced with a real type in the .NET type system at runtime.

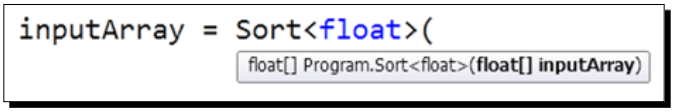
So once we have this method, we can call it as follows:

```
int[] inputArray = { 1, 2, 0, 3 };
inputArray = Sort<int>(inputArray);
```

However, if you hover your mouse pointer right after the first brace (`(`), you can see in the tooltip, the expected type is already `int[]`, as shown in the following screenshot:

That's the beauty of Generics. As we had mentioned `int` inside `<` and `>`, the compiler now knows for sure that it should expect only an `int[]` as the argument to the `Sort<T> ()` method.

However, if you change `int` to `float`, you will see that the expectation of the compiler also changes. It then expects a `float[]` as the argument, as shown:



Now if you think you can fool the compiler by passing an integer array while it is asking for a float, you are wrong. That's blocked by compiler-time type checking. If you try something similar to the following:

```
int[] inputArray = { 1, 2, 0, 3 };
inputArray = Sort<float>(inputArray);
```

You will get the following compiler error:

```
Argument 1: cannot convert from 'int[]' to 'float[]'
```

This means that Generics ensures strong type safety and is an integral part of the .NET framework, which is type safe.

Reason 2: Generics can save you type safety woes, big time

The previous example was about a sorting algorithm that doesn't change with data type. There are other things that become easier while dealing with Generics.

There are broadly two types of operations that can be performed on a list of elements:

1. Location centric operations
2. Data centric operations

Adding some elements at the front and deleting elements at an index are a couple of examples of location-centric operations on a list of data. In such operations, the user doesn't need to know about the data. It's just some memory manipulation at best.

However, if the request is to delete every odd number from a list of integers, then that's a data-centric operation. To be able to successfully process this request, the method has to know how to determine whether an integer is odd or not. This might sound trivial for an integer; however, the point is the logic of determining whether an element is a candidate for deletion or not, is not readily known to the compiler. It has to be delegated.

Before Generics appeared in .NET 2.0, people were using (and unfortunately these are still in heavy use) non-generic collections that are capable of storing a list of objects.

As an object sits at the top of the hierarchy in the .NET object model, this opens floodgates. If such a list exists and is exposed, people can put in just about anything in that list and the compiler won't complain a bit, because to the compiler everything is fine as they are all objects.

So, if a loosely typed collection such as `ArrayList` is used to store objects of type `T`, then for any data-centric operation, these must be down-casted to `T` again. Now, if somehow an entry that is not `T`, is put into the list, then this down-casting will result in an exception at runtime.

Suppose, I want to maintain a list of my students, then we can do that by using `ArrayList` to store a list of such `Student` objects:

```
class Student
{
    public char Grade
    {
        get; set;
    }

    public int Roll
    {
        get; set;
    }

    public string Name
    {
        get; set;
    }
}

//List of students
ArrayList studentList = new ArrayList();

Student newStudent = new Student();
newStudent.Name = "Dorothy";
newStudent.Roll = 1;
newStudent.Grade = 'A';

studentList.Add(newStudent);

newStudent = new Student();
newStudent.Name = "Sam";
newStudent.Roll  = 2;
newStudent.Grade ='B';

studentList.Add(newStudent);

foreach (Object s in studentList)
{
    //Type-casting. If s is anything other than a student
```

```
        //or a derived class, this line will throw an exception.
        //This is a data centric operation.
        Student currentStudent = (Student)s;
        Console.WriteLine("Roll # " + currentStudent.Roll + " " +
                          currentStudent.Name + " Scored a " +
                          curr entStudent.Grade);
    }
```

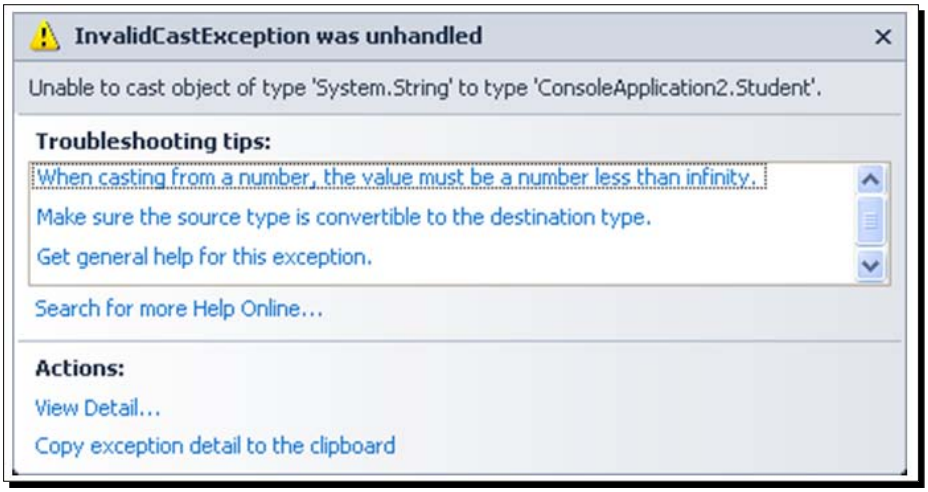
What's the problem with this approach?

All this might look kind of okay, because we have been taking great care not to put anything else in the list other than `Student` objects. So, while we de-reference them after boxing, we don't see any problem. However, as the `ArrayList` can take any object as the argument, we could, by mistake, write something similar to the following:

```
studentList.Add("Generics"); //Fooling the compiler
```

As `ArrayList` is a loosely typed collection, it doesn't ensure *compile-time type checking*. So, this code won't generate any compile-time warning, and eventually it will throw the following exception at runtime when we try to de-reference this, to put in a `Student` object.

Then, it will throw an `InvalidCastException`:



What the exception in the preceding screenshot actually tells us is that Generics is a string and it can't cast that to `Student`, for the obvious reason that the compiler has no clue how to convert a string to a `Student` object.

Unfortunately, this only gets noticed by the compiler during runtime. With Generics, we can catch this sort of error early on at compile time.

Following is the generic code to maintain that list:

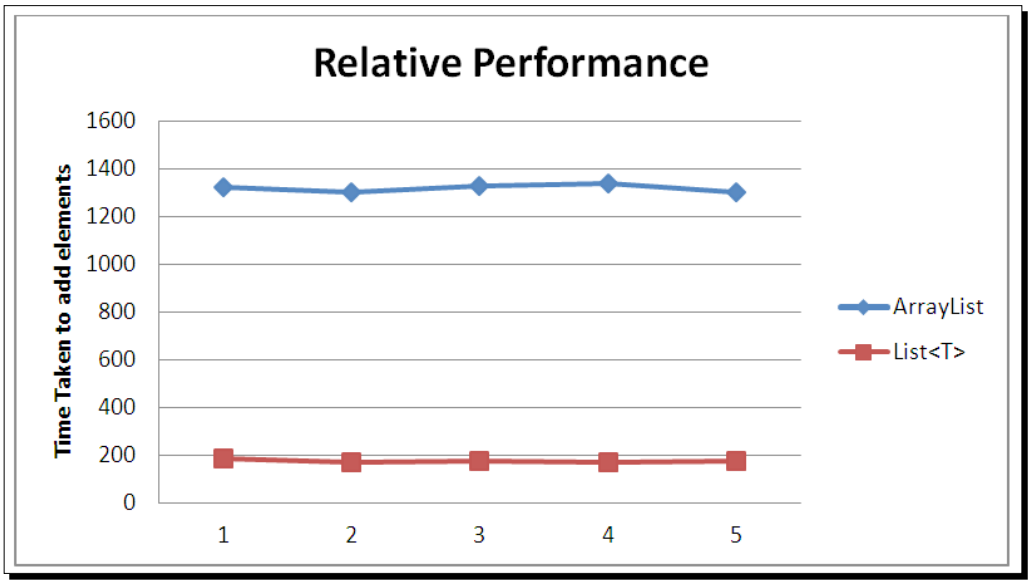
```
//Creating a generic list of type "Student".
//This is a strongly-typed-collection of type "Student".
//So nothing, except Student or derived class objects from Student
//can be put in this list myStudents
List<Student> myStudents = new List<Student>();

//Adding a couple of students to the list
Student newStudent = new Student();
newStudent.Name = "Dorothy";
newStudent.Roll = 1;
newStudent.Grade = 'A';

myStudents.Add(newStudent);

newStudent = new Student();
newStudent.Name = "Sam";
newStudent.Roll = 2;
newStudent.Grade = 'B';

myStudents.Add(newStudent);

//Looping through the list of students
foreach (Student currentStudent in myStudents)
{
      //There is no need to type cast. Because compiler
      //already knows that everything inside this list
      //is a Student.
      Console.WriteLine("Roll # " + currentStudent.Roll + " " +
                        currentStudent.Name + " Scored a " +
                        currentStudent.Grade);
}
```

The reasons mentioned earlier are the basic benefits of Generics. Also with Generics, language features such as LINQ and completely new languages such as F# came into existence. So, this is important. I hope you are convinced that Generics is a great programming tool and you are ready to learn it.

Reason 3: Generics leads to faster code

In the .NET Framework, everything is an object so it's okay to throw in anything to the non-generic loosely typed collection such as `ArrayList`, as shown in the previous example. This means we have to box (up-cast to object for storing things in the `Arraylist`; this process is implicit) and unbox (down-cast the object to the desired object type). This leads to slower code.

Here is the result of an experiment. I created two lists, one `ArrayList` and one `List<int>` to store integers:



And following is the data that drove the preceding graph:

ArrayList	List<T>
1323	185
1303	169
1327	172
1340	169
1302	172

The previous table mentions the total time taken in milliseconds to add 10,000,000 elements to the list. Clearly, generic collection is about seven times faster.

Reason 4: Generics is now ubiquitous in the .NET ecosystem

Look around. If you care to develop any non-trivial application, you are better off using some of the APIs built for the specific job at hand. Most of the APIs available rely heavily on strong typing and they achieve this through Generics. We shall discuss some of these APIs (LINQ, PowerCollections, C5) that are being predominantly used by the .NET community in this book.

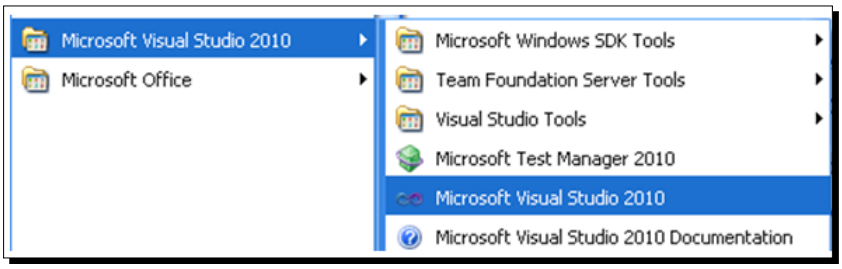
So far, I have been giving you reasons to learn Generics. At this point, I am sure, you are ready to experiment with .NET Generics. Please check out the instructions in the next section to install the necessary software if you don't have it already.

Setting up the environment

If you are already running any 2010 version of Visual Studio that lets you create C# windows and console projects, you don't have to do anything and you can skip this section.

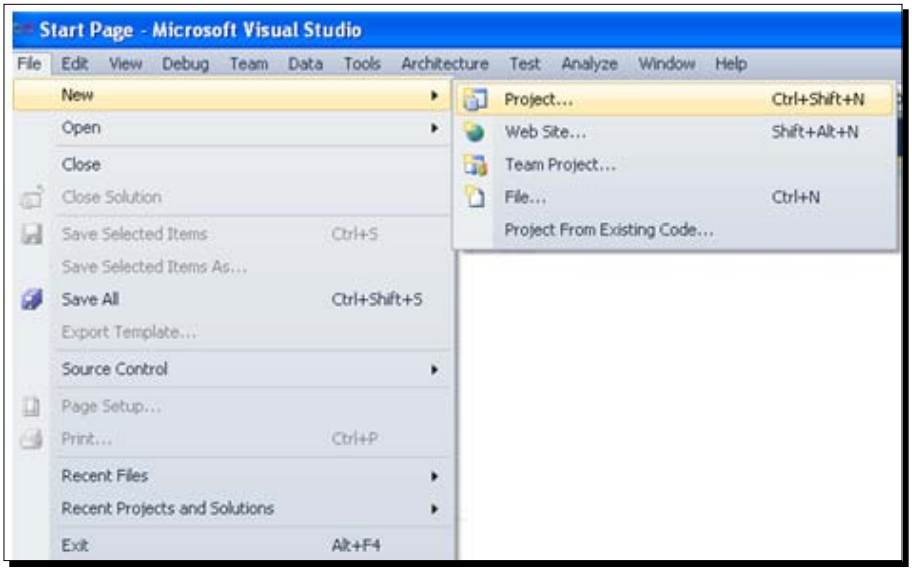
You can download and install the Visual Studio Trial from `http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=12752`.

Once you are done, you should see the following in your program menu:

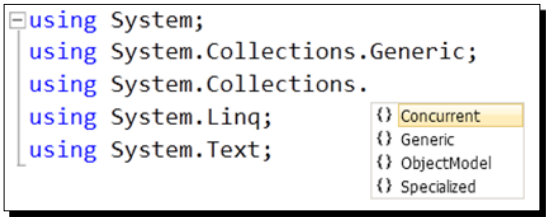


After this, start the program highlighted in the preceding screenshot **Microsoft Visual Studio 2010**.

Then go to the **File** menu to create a console project:



Now, once that is created, make sure the following namespaces are available:



If these are available, you have done the right setup. Congratulations!

Summary

My objective for this chapter was to make sure you get why Generics is important. Following are the points again in bullets:

- It ensures compile-time type checking, so type safety is ensured.
- It can yield the right code for the data type thrown at it at runtime, thus saving us a lot of typing.
- It is very fast (about seven times) compared to its non-generic cousins for value types.
- It is everywhere in the .NET ecosystem. API/framework developers trust the element of least surprise and they know people are familiar with Generics and their syntax. So they try to make sure their APIs also seem familiar to the users.

In the end, we did an initial setup of the environment; so we are ready to build and run applications using .NET Generics. From the next chapter, we shall learn about .NET Generic containers and classes. In the next chapter, we shall discuss the Generic container `List<T>` that will let you store any type of data in a type safe way. Now that you know that's important, let's go there.

2
Lists

Don't forget the milk, honey!

Lists are everywhere, starting from the laundry list and grocery list to the checklist on your smartphone's task manager. There are two types of lists. Simple lists, which just store some items disregarding the order allowing duplicates; and other types which don't allow duplicates. These second types of lists which don't allow duplicates are called sets in mathematics. The other broad category of list is associative list, where each element in some list gets mapped to another element in another list.

In this chapter, we shall learn about these generic containers and related methods. We shall learn when to use which list-based container depending on the situation.

Reading this chapter and following the exercises you will learn the following:

- Why bother learning about generic lists?
- Types of generic lists and how to use them