12

Generics

\*\*\*COMP: Insert “Begin 2.0” tab. Note that there should be “2.0” tabs on every page of this chapter.

|  |
| --- |
| A |

s your projects become more sophisticated, you will need a better way to reuse and customize existing software. To facilitate code reuse, especially the reuse of algorithms, C# includes a feature called generics.

Generics are lexically like generic types in Java and templates in C++. In all three languages, these features enable the implementation of algorithms and patterns once, rather than requiring separate implementations for each type that the algorithm or pattern operates on. However, C# generics are very different from both Java generics and C++ templates in the details of their implementation and impact on the type system of their respective languages.

\*\*\*COMP: Insert 12mindmap pick up from previous edition 9781509303588 p. 487

|  |
| --- |
| Macintosh HD:Users:annapopick:Desktop:Freelance:Pearson Freelance:Pearson_InProgress:9781509303588_Michaelis:Michaelis_Author:Michaelis_Word_AllEdits:Michaelis_Art:Michaelis_Mindmaps:12mindmap.pdf |

Generics were added to the runtime and C# in version 2.0.

C# without Generics

We begin the discussion of generics by examining a class that does not use generics. This class, System.Collections.Stack, represents a collection of objects such that the last item to be added to the collection is the first item retrieved from the collection (last in, first out [LIFO]). Push() and Pop(), the two main methods of the Stack class, add items to the stack and remove them from the stack, respectively. The declarations for the methods on the Stack class appear in Listing 12.1.

Listing 12.1: The System.Collections.Stack Method Signatures

public class Stack

{

public virtual object Pop() { ... }

public virtual void Push(object obj) { ... }

// ...

}

Programs frequently use stack type collections to facilitate multiple undo operations. For example, Listing 12.2 uses the System.Collections.Stack class for undo operations within a program that simulates the Etch A Sketch game.

Listing 12.2: Supporting Undo in a Program Similar to the Etch A Sketch Game

using System.Collections;

class Program

{

// ...

public void Sketch()

{

Stack path = new Stack();

Cell currentPosition;

ConsoleKeyInfo key; // Added in C# 2.0

do

{

// Etch in the direction indicated by the

// arrow keys that the user enters

key = Move();

switch (key.Key)

{

case ConsoleKey.Z:

// Undo the previous Move

if (path.Count >= 1)

{

currentPosition = (Cell)path.Pop();

Console.SetCursorPosition(

currentPosition.X, currentPosition.Y);

Undo();

}

break;

case ConsoleKey.DownArrow:

case ConsoleKey.UpArrow:

case ConsoleKey.LeftArrow:

case ConsoleKey.RightArrow:

// SaveState()

currentPosition = new Cell(

Console.CursorLeft, Console.CursorTop);

path.Push(currentPosition);

break;

default:

Console.Beep(); // Added in C# 2.0

break;

}

}

while (key.Key != ConsoleKey.X); // Use X to quit

}

}

public struct Cell

{

// Use read-only field prior to C# 6.0

public int X { get; }

public int Y { get; }

public Cell(int x, int y)

{

X = x;

Y = y;

}

}

The results of Listing 12.2 appear in Output 12.1.

Output 12.1

\*\*\*COMP: Insert 12output01 pick up figure 12.1 from previous edition 9781509303588 p. 490

|  |
| --- |
| Macintosh HD:Users:annapopick:Desktop:Freelance:Pearson Freelance:Pearson_InProgress:9781509303588_Michaelis:Michaelis_Author:Michaelis_Word_AllEdits:Michaelis_Art:Michaelis_Outputs:12output01.tif |

Using the variable path, which is declared as a System.Collections.Stack, you save the previous move by passing a custom type, Cell, into the Stack.Push() method using path.Push(currentPosition). If the user enters a Z (or presses Ctrl+Z), you undo the previous move by retrieving it from the stack using a Pop() method, setting the cursor position to be the previous position, and calling Undo().

Although this code is functional, the System.Collections.Stack class has a fundamental shortcoming. As shown in Listing 12.1, the Stack class collects values of type object. Because every object in the Common Language Runtime (CLR) derives from object, Stack provides no validation that the elements you place into it are homogenous or are of the intended type. For example, instead of passing currentPosition, you can pass a string in which X and Y are concatenated with a decimal point between them. However, the compiler must allow the inconsistent data types because the stack class is written to take any object, regardless of its more specific type.

Furthermore, when retrieving the data from the stack using the Pop() method, you must cast the return value to a Cell. But if the type of the value returned from the Pop() method is not Cell, an exception is thrown. By deferring type checking until runtime by using a cast, you make the program more brittle. The fundamental problem with creating classes that can work with multiple data types without generics is that they must work with a common base class (or interface), usually object.

Using value types, such as a struct or an integer, with classes that use object exacerbates the problem. If you pass a value type to the Stack.Push() method, for example, the runtime automatically boxes it. Similarly, when you retrieve a value type, you need to explicitly unbox the data and cast the object reference you obtain from the Pop() method into a value type. Casting a reference type to a base class or interface has a negligible performance impact, but the box operation for a value type introduces more overhead, because it must allocate memory, copy the value, and then later garbage-collect that memory.

C# is a language that encourages type safety: The language is designed so that many type errors, such as assigning an integer to a variable of type string, can be caught at compile time. The fundamental problem is that the Stack class is not as type-safe as one expects a C# program to be. To change the Stack class to enforce type safety by restricting the contents of the stack to be a particular data type (without using generic types), you must create a specialized stack class, as in Listing 12.3.

Listing 12.3: Defining a Specialized Stack Class

public class CellStack

{

public virtual Cell Pop();

public virtual void Push(Cell cell);

// ...

}

Because CellStack can store only objects of type Cell, this solution requires a custom implementation of the stack methods, which is less than ideal. Implementing a type-safe stack of integers would require yet another custom implementation; each implementation would look remarkably like every other one. There would be lots of duplicated, redundant code.

|  |  |
| --- | --- |
|  | |
|  | Beginner Topic  Another Example: Nullable Value Types  Chapter 3 introduced the capability of declaring variables that could contain null by using the nullable modifier, ?, when declaring a value type variable. C# began supporting this functionality only in version 2.0, because the right implementation required generics. Prior to the introduction of generics, programmers faced essentially two options.  The first option was to declare a nullable data type for each value type that needs to handle null values, as shown in Listing 12.4.  Listing 12.4: Declaring Versions of Various Value Types That Store null  struct NullableInt  {  /// <summary>  /// Provides the value when HasValue returns true  /// </summary>  public int Value{ get; private set; }  /// <summary>  /// Indicates whether there is a value or whether  /// the value is "null"  /// </summary>  public bool HasValue{ get; private set; }  // ...  }  struct NullableGuid  {  /// <summary>  /// Provides the value when HasValue returns true  /// </summary>  public Guid Value{ get; private set; }  /// <summary>  /// Indicates whether there is a value or whether  /// the value is "null"  /// </summary>  public bool HasValue{ get; private set; }  ...  }  ...  Listing 12.4 shows possible implementations of NullableInt and NullableGuid. If a program required additional nullable value types, you would have to create yet another struct with the properties modified to use the desired value type. Any improvement of the implementation (e.g., adding a user-defined implicit conversion from the underlying type to the nullable type) would require modifying all the nullable type declarations.  An alternative strategy for implementing a nullable type without generics is to declare a single type with a Value property of type object, as shown in Listing 12.5.  Listing 12.5: Declaring a Nullable Type That Contains a Value Property of Type object  struct Nullable  {  /// <summary>  /// Provides the value when HasValue returns true  /// </summary>  public object Value{ get; private set; }  /// <summary>  /// Indicates whether there is a value or whether  /// the value is "null"  /// </summary>  public bool HasValue{ get; private set; }  ...  }  Although this option requires only one implementation of a nullable type, the runtime always boxes value types when setting the Value property. Furthermore, retrieving the underlying value from the Value property requires a cast operation, which might potentially be invalid at runtime.  Neither option is particularly attractive. To eliminate this problem, C# 2.0 introduced generics to C#. (And, in fact, nullable types are implemented as the generic type Nullable<T>.) |
|  | |

Introducing Generic Types

Generics provide a facility for creating data structures that can be specialized to handle specific types. Programmers define these parameterized types so that each variable of a particular generic type has the same internal algorithm, but the types of data and method signatures can vary on the basis of the type arguments provided for the type parameters.

To minimize the learning curve for developers, the C# designers chose syntax that superficially resembles C++ templates. In C#, the syntax for generic classes and structures uses angle brackets to both declare the generic type parameters in the type declaration and specify the generic type arguments when the type is used.

Using a Generic Class

Listing 12.6 shows how you can specify the actual type argument used by the generic class. You instruct the path variable to be the “Stack of Cell” type by specifying Cell within angle bracket notation in both the object creation expression and the declaration statement. In other words, when declaring a variable (path in this case) using a generic data type, C# requires the developer to identify the actual type arguments used by the generic type. Listing 12.6 illustrates this process with the new generic Stack class.

Listing 12.6: Implementing Undo with a Generic Stack Class

using System;

using System.Collections.Generic;

class Program

{

// ...

public void Sketch()

{

Stack<Cell> path; // Generic variable declaration

path = new Stack<Cell>(); // Generic object instantiation

Cell currentPosition;

ConsoleKeyInfo key;

do

{

// Etch in the direction indicated by the

// arrow keys entered by the user

key = Move();

switch (key.Key)

{

case ConsoleKey.Z:

// Undo the previous Move

if (path.Count >= 1)

{

// No cast required

currentPosition = path.Pop();

Console.SetCursorPosition(

currentPosition.X, currentPosition.Y);

Undo();

}

break;

case ConsoleKey.DownArrow:

case ConsoleKey.UpArrow:

case ConsoleKey.LeftArrow:

case ConsoleKey.RightArrow:

// SaveState()

currentPosition = new Cell(

Console.CursorLeft, Console.CursorTop);

// Only type Cell allowed in call to Push()

path.Push(currentPosition);

break;

default:

Console.Beep(); // Added in C# 2.0

break;

}

} while (key.Key != ConsoleKey.X); // Use X to quit

}

}

The results of Listing 12.6 appear in Output 12.2.

Output 12.2

\*\*\*COMP: Insert 12output02 pick up figure 12.2 from previous edition 9781509303588 p. 495

|  |
| --- |
| Macintosh HD:Users:annapopick:Desktop:Freelance:Pearson Freelance:Pearson_InProgress:9781509303588_Michaelis:Michaelis_Author:Michaelis_Word_AllEdits:Michaelis_Art:Michaelis_Outputs:12output02.tif |

In the path declaration shown in Listing 12.6, you declare a variable and initialize it with a new instance of the System.Collections.Generic.Stack<Cell> class. You specify in angle brackets that the data type of the stack’s elements is Cell. As a result, every object added to and retrieved from path is of type Cell. In turn, you no longer need to cast the return of path.Pop() or ensure that only Cell type objects are added to path in the Push() method.

Defining a Simple Generic Class

Generics allow you to author algorithms and patterns and to reuse the code for different data types. Listing 12.7 creates a generic Stack<T> class similar to the System.Collections.Generic.Stack<T> class used in the code in Listing 12.6. You specify a type parameter (in this case, T) within angle brackets after the class name. The generic Stack<T> can then be supplied with a single type argument that is substituted everywhere T appears in the class. Thus, the stack can store items of any stated type, without duplicating code or converting the item to type object. The type parameter T is a placeholder that must be supplied with a type argument. In Listing 12.7, you can see that the type parameter will be used for the internal Items array, the type for the parameter to the Push() method, and the return type for the Pop() method.

Listing 12.7: Declaring a Generic Class, Stack<T>

public class Stack<T>

{

public Stack(int maxSize)

{

InternalItems = new T[maxSize];

}

// Use read-only field prior to C# 6.0

private T[] InternalItems { get; }

public void Push(T data)

{

...

}

public T Pop()

{

...

}

}

Benefits of Generics

There are several advantages of using a generic class rather than a nongeneric version (such as the System.Collections.Generic.Stack<T> class used earlier instead of the original System.Collections.Stack type):

1. Generics facilitate increased type safety, preventing data types other than those explicitly intended by the members within the parameterized class. In Listing 12.7, the parameterized stack class restricts you to the Cell data type when using Stack<Cell>. For example, the statement path.Push("garbage") produces a compile-time error indicating that there is no overloaded method for System.Collections.Generic.Stack<T>.Push(T) that can work with the string, because it cannot be converted to a Cell.

2. Compile-time type checking reduces the likelihood of InvalidCastException type errors at runtime.

3. Using value types with generic class members no longer causes a boxing conversion to object. For example, path.Pop() and path.Push() do not require an item to be boxed when added or unboxed when removed.

4. Generics in C# reduce code bloat. Generic types retain the benefits of specific class versions, without the overhead. For example, it is no longer necessary to define a class such as CellStack.

5. Performance improves because casting from an object is no longer required, thereby eliminating a type check operation. Also, performance improves because boxing is no longer necessary for value types.

6. Generics reduce memory consumption by avoiding boxing and, therefore, consuming less memory on the heap.

7. Code becomes more readable because of fewer casting checks and because of the need for fewer type-specific implementations.

8. Editors that assist coding via some type of IntelliSense work directly with return parameters from generic classes. There is no need to cast the return data for IntelliSense to work.

At their core, generics offer the ability to code pattern implementations and then reuse those implementations wherever the patterns appear. Patterns describe problems that occur repeatedly within code, and templates provide a single implementation for these repeating patterns.

Type Parameter Naming Guidelines

Just as when naming a method’s formal parameter, so you should be as descriptive as possible when naming a type parameter. Furthermore, to distinguish the parameter as being a type parameter, its name should include a T prefix. For example, in defining a class such as EntityCollection<TEntity>, you would use the type parameter name TEntity.

The only time you would not use a descriptive type parameter name is when such a description would not add any value. For example, using T in the Stack<T> class is appropriate, since the indication that T is a type parameter is sufficiently descriptive; the stack works for any type.

In the next section, you will learn about constraints. It is a good practice to use constraint-descriptive type names. For example, if a type parameter must implement IComponent, consider using a type name of TComponent.

|  |
| --- |
| Guidelines  DO choose meaningful names for type parameters and prefix the namjhjjjhhjjjhhjjhe with Tlhlhkjh.  CONSIDER indicating a constraint in the name of a type parameter. |

Generic Interfaces and Structs

C# supports the use of generics throughout the language, including interfaces and structs. The syntax is identical to that used by classes. To declare an interface with a type parameter, place the type parameter in angle brackets immediately after the interface name, as shown in the example of IPair<T> in Listing 12.8.

Listing 12.8: Declaring a Generic Interface

interface IPair<T>

{

T First { get; set; }

T Second { get; set; }

}

This interface represents pairs of like objects, such as the coordinates of a point, a person’s genetic parents, or nodes of a binary tree. The type contained in the pair is the same for both items.

To implement the interface, you use the same syntax as you would for a nongeneric class. Note that it is legal—indeed, common—for the type argument for one generic type to be a type parameter of another generic type, as shown in Listing 12.9. The type argument of the interface is the type parameter declared by the class. In addition, this example uses a struct rather than a class, demonstrating that C# supports custom generic value types.

Listing 12.9: Implementing a Generic Interface

public struct Pair<T>: IPair<T>

{

public T First { get; set; }

public T Second { get; set; }

}

Support for generic interfaces is especially important for collection classes, where generics are most prevalent. Before generics were available in C#, developers relied on a series of interfaces within the System.Collections namespace. Like their implementing classes, these interfaces worked only with type object, and as a result, the interface forced all access to and from these collection classes to require a cast. By using type-safe generic interfaces, you can avoid cast operations.

\*\*\*COMP: End Advanced Topic after Guidelines

|  |  |
| --- | --- |
|  | |
|  | AdVanced Topic  Implementing the Same Interface Multiple Times on a Single Class  Two different constructions of the same generic interface are considered different types. Consequently, “the same” generic interface can be implemented multiple times by a class or struct. Consider the example in Listing 12.10.  Listing 12.10: Duplicating an Interface Implementation on a Single Class  public interface IContainer<T>  {  ICollection<T> Items { get; set; }  }  public class Person: IContainer<Address>,  IContainer<Phone>, IContainer<Email>  {  ICollection<Address> IContainer<Address>.Items  {  get{...}  set{...}  }  ICollection<Phone> IContainer<Phone>.Items  {  get{...}  set{...}  }  ICollection<Email> IContainer<Email>.Items  {  get{...}  set{...}  }  }  In this example, the Items property appears multiple times using an explicit interface implementation with a varying type parameter. Without generics, this would not be possible; instead, the compiler would allow only one explicit IContainer.Items property.  However, this technique of implementing multiple versions of the same interface is considered by many developers to be a “bad code smell” because it is potentially confusing (particularly if the interface permits covariant or contravariant conversions). Moreover, the Person class here seems potentially badly designed; one does not normally think of a person as being “a thing that can provide a set of email addresses.” When you feel tempted to make a class implement three versions of the same interface, consider whether it might be better to make it instead implement three properties—for example, EmailAddresses, PhoneNumbers, and MailingAddresses—each of which returns the appropriate construction of the generic interface.   |  | | --- | | Guidelines  AVOID implementing multiple constructions of the same generic interface in one type. | |
|  | |

Defining a Constructor and a Finalizer

Perhaps surprisingly, the constructors (and finalizer) of a generic class or struct do not require type parameters; in other words, they do not require Pair<T>(){...}. In the pair example in Listing 12.11, the constructor is declared using public Pair(T first, T second).

Listing 12.11: Declaring a Generic Type’s Constructor

public struct Pair<T>: IPair<T>

{

public Pair(T first, T second)

{

First = first;

Second = second;

}

public T First { get; set; }

public T Second { get; set; }

}

Specifying a Default Value with the default operator

Listing 12.11 included a constructor that takes the initial values for both First and Second and assigns them to First and Second. Since Pair<T> is a struct, any constructor you provide must initialize all fields and automatically implemented properties. This presents a problem, however.

Consider a constructor for Pair<T> that initializes only half of the pair at instantiation time. Defining such a constructor, as shown in Listing 12.12, causes a compile-time error because the field Second is still uninitialized at the end of the constructor. Providing initialization for Second presents a problem because you don’t know the data type of T. If it is a nullable type, null would work, but this approach would not work if T were a non-nullable type.

Listing 12.12: Not Initializing All Fields, Causing a Compile-Time Error

public struct Pair<T>: IPair<T>

{

// ERROR: Field 'Pair<T>.Second' must be fully assigned

// before control leaves the constructor

// public Pair(T first)

// {

// First = first;

// }

// ...

}

\*\*\*COMP: Insert “Begin 7.0” tab

To deal with this scenario, C# provides the default operator. The default value of int, for example, could be specified with default (assuming C# 7.1). In the case of T, which Second requires, you can use default, as shown in Listing 12.13.

Listing 12.13: Initializing a Field with the default Operator

public struct Pair<T>: IPair<T>

{

public Pair(T first)

{

First = first;

Second = default;

}

// ...

}

The default operator can provide the default value for any type, including type parameters.

Prior to C# 7.1, it was necessary to always to pass a type parameter to the default operator, as in Second = default(T). However, C# 7.1 includes the option to use default without specifying a parameter if it is possible to infer the data type. For example, with variable initialization or assignment, you can use Pair<T> pair = default in place of Pair<T> pair = default(Pair<T>). Furthermore, if a method returns an int, it is possible to simply use return default and have the compiler infer a default(int) from the return of the method. Other scenarios where such inference is possible are default parameter (optional) values and method call arguments.

\*\*\*COMP: Insert “Begin 8.0” tab

Note that all nullable types have null as their default value, as do nullable generic types such as default(T?). Furthermore, the default value for all reference types is null. As a result, if nullable reference types are enabled with C# 8.0, assigning default to a non-nullable reference type will result in a warning. Unfortunately, any code that assigns default to a reference type in C# 7.0 or earlier will, therefore, produce a warning when being upgraded to support nullability in C# 8.0. For this reason, prior to C# 8.0 (and obviously after it), you should avoid assigning default or null to reference types unless null is expected to be a valid value. Where possible, favor leaving the variable uninitialized until a valid value is available for assignment. In cases like Listing 12.13, where an appropriate value for Second is unknown in the constructor, Second will inevitably be null for a reference type or a nullable value type; thus, a warning appears when assigning default (which is potentially null) to a property of generic type T. To appropriately handle the warning, you will need to declare the Second property to be of type T? and identify whether T is a reference type or a value type with a class/struct constraint as described in the “struct/class Constraints” section. (In fact, this leads to the more general guideline not to assign default on a generic type unless the type assigned is constrained to a class or a struct.)

\*\*\*COMP: Insert “End 8.0” tab

\*\*\*COMP: Insert “End 7.0” tab

Multiple Type Parameters

Generic types may declare any number of type parameters. The initial Pair<T> example contained only one type parameter. To enable support for storing a dichotomous pair of objects, such as a name/value pair, you could create a new version of the type that declares two type parameters, as shown in Listing 12.14.

Listing 12.14: Declaring a Generic with Multiple Type Parameters

interface IPair<TFirst, TSecond>

{

TFirst First { get; set; }

TSecond Second { get; set; }

}

public struct Pair<TFirst, TSecond>: IPair<TFirst, TSecond>

{

public Pair(TFirst first, TSecond second)

{

First = first;

Second = second;

}

public TFirst First { get; set; }

public TSecond Second { get; set; }

}

When you use the Pair<TFirst, TSecond> class, you supply multiple type parameters within the angle brackets of the declaration and instantiation statements; you then supply matching types to the parameters of the methods when you call them. Listing 12.15 illustrates this approach.

Listing 12.15: Using a Type with Multiple Type Parameters

Pair<int, string> historicalEvent =

new Pair<int, string>(1914,

"Shackleton leaves for South Pole on ship Endurance");

Console.WriteLine("{0}: {1}",

historicalEvent.First, historicalEvent.Second);

The number of type parameters—that is, the arity—uniquely distinguishes the class from others of the same name. Because of this arity variation, it is possible to define both Pair<T> and Pair<TFirst, TSecond> within the same namespace. Furthermore, because of their close semantic relationship, generics that differ only by arity should be placed into the same C# file.

|  |
| --- |
| Guidelines  DO place multiple generic semantically equivalent classes into a single file if they differ only by the number of generic parameters. |

\*\*\*COMP: Insert “Begin 7.0” tab

|  |  |
| --- | --- |
|  | |
|  | Beginner Topic  Tuples: An Abundance of Arity  We introduced C# 7.0’s support for tuple syntax back in Chapter 3. Internally, the underlying type that implements the tuple syntax is, in fact, a generic—specifically a System.ValueTuple. As with Pair<...>, it is possible to reuse the same name because of the variation in arity (each class has a different number of type parameters), as shown in Listing 12.16.  Listing 12.16: Using Arity to Overload a Type Definition  public class ValueTuple { ... }  public class ValueTuple<T1>:  IStructuralEquatable, IStructuralComparable, IComparable {...}  public class ValueTuple<T1, T2>: ... {...}  public class ValueTuple<T1, T2, T3>: ... {...}  public class ValueTuple<T1, T2, T3, T4>: ... {...}  public class ValueTuple<T1, T2, T3, T4, T5>: ... {...}  public class ValueTuple<T1, T2, T3, T4, T5, T6>: ... {...}  public class ValueTuple<T1, T2, T3, T4, T5, T6, T7>: ... {...}  public class ValueTuple<T1, T2, T3, T4, T5, T6, T7, TRest>: ... {...}  The ValueTuple<...> set of classes was designed for the same purpose as the Pair<T> and Pair<TFirst, TSecond> classes, except that together they can handle eight type arguments. In fact, using the last ValueTuple shown in Listing 12.16, TRest can be used to store another ValueTuple, making the number of elements of the tuple practically unlimited. And, if you define such a tuple using C# 7.0’s tuple syntax, that is what the compiler will generate.  Another interesting member of the tuple family of classes is the nongeneric ValueTuple class. This class has eight static factory methods for instantiating the various generic tuple types. Although each generic type could be instantiated directly using its constructor, the ValueTuple type’s factory methods allow for inference of the type arguments via the Create() method. This is not important in C# 7.0 because the code is as simple as var keyValuePair = ("555-55-5555", new Contact("Inigo Montoya")) (assuming no named items). However, as shown in Listing 12.17, using the Create() method in combination with type inference is simpler for C# 6.0.  Listing 12.17: Comparing System.ValueTuple Instantiation Approaches  #if !PRECSHARP7  (string, Contact) keyValuePair;  keyValuePair =  ("555-55-5555", new Contact("Inigo Montoya"));  #else // Use System.ValueTupe<string,Contact> prior to C# 7.0  ValueTuple<string, Contact> keyValuePair;  keyValuePair =  ValueTuple.Create(  "555-55-5555", new Contact("Inigo Montoya"));  keyValuePair =  new ValueTuple<string, Contact>(  "555-55-5555", new Contact("Inigo Montoya"));  #endif // !PRECSHARP7  Obviously, when the ValueTuple gets large, the number of type parameters to specify could be cumbersome without the Create() factory methods.  \*\*\*COMP: Insert “Begin 4.0” tab  Note that a similar tuple class was added in C# 4.0: System.Tuple. However, it was determined that abundant use of C# 7.0’s tuple syntax and the resulting prevalence of tuples it would introduce warranted creating the System.ValueTuple type because of the performance improvements it provided.  \*\*\*COMP: Insert “End 4.0” tab  \*\*\*COMP: Insert “End 7.0” tab |
|  | |

As you might have deduced from the fact that the framework libraries declare eight different generic System.ValueTuple types, there is no support in the CLR type system for variadic generic types. Methods can take an arbitrary number of arguments by using parameter arrays, but there is no corresponding technique for generic types; every generic type must be of a specific arity. (See Beginner Topic: Tuples: An Abundance of Arity for an example where you might imagine such a feature.)

Nested Generic Types

Type parameters on a containing generic type will “cascade” down to any nested types automatically. If the containing type declares a type parameter T, for example, all nested types will also be generic, and type parameter T will be available on the nested type as well. If the nested type includes its own type parameter named T, this will hide the type parameter within the containing type, and any reference to T in the nested type will refer to the nested T type parameter. Fortunately, reuse of the same type parameter name within the nested type will cause a compiler warning to prevent accidental overlap (see Listing 12.18).

Listing 12.18: Nested Generic Types

class Container<T1, T2>

{

// Nested classes inherit type parameters.

// Reusing a type parameter name will cause

// a warning.

class Nested<T2>

{

void Method(T1 param0, T2 param1)

{

}

}

}

The containing type’s type parameters are accessible in the nested type in the same way that members of the containing type are also accessible from the nested type. The rule is simply that a type parameter is available anywhere within the body of the type that declares it.

|  |
| --- |
| Guidelines  AVOID shadowing a type parameter of an outer type with an identically named type parameter of a nested type. |

Constraints

Generics support the ability to define constraints on type parameters. These constraints ensure that the types provided as type arguments conform to various rules. Take, for example, the BinaryTree<T> class shown in Listing 12.19.

Listing 12.19: Declaring a BinaryTree<T> Class with No Constraints

public class BinaryTree<T>

{

public BinaryTree ( T item)

{

Item = item;

}

public T Item { get; set; }

public Pair<BinaryTree<T>> SubItems { get; set; }

}

(An interesting side note is that BinaryTree<T> uses Pair<T> internally, which is possible because Pair<T> is simply another type.)

Suppose you want the tree to sort the values within the Pair<T> value as it is assigned to the SubItems property. To achieve the sorting, the SubItems set accessor uses the CompareTo() method of the supplied key, as shown in Listing 12.20.

Listing 12.20: Needing the Type Parameter to Support an Interface

public class BinaryTree<T>

{

public BinaryTree(T item)

{

Item = item;

}

public T Item { get; set; }

public Pair<BinaryTree<T>> SubItems

{

get{ return \_SubItems; }

set

{

IComparable<T> first;

// ERROR: Cannot implicitly convert type...

first = value.First; // Explicit cast required

if (first.CompareTo(value.Second) < 0)

{

// first is less than second

// ...

}

else

{

// first and second are the same or

// second is less than first

// ...

}

\_SubItems = value;

}

}

private Pair<BinaryTree<T>> \_SubItems;

}

At compile time, the type parameter T is an unconstrained generic. When the code is written as shown in Listing 12.20, the compiler assumes that the only members available on T are those inherited from the base type object, since every type has object as a base class. Only methods such as ToString(), therefore, are available to call on an instance of the type parameter T. As a result, the compiler displays a compilation error because the CompareTo() method is not defined on type object.

You can cast the T parameter to the IComparable<T> interface to access the CompareTo() method, as shown in Listing 12.21.

Listing 12.21: Needing the Type Parameter to Support an Interface or Exception Thrown

public class BinaryTree<T>

{

public BinaryTree(T item)

{

Item = item;

}

public T Item { get; set; }

public Pair<BinaryTree<T>?>? SubItems

{

get{ return \_SubItems; }

set

{

switch(value)

{

// Null handling removed for elucidation

// Using C# 8.0 Pattern Matching. Switch to

// checking for null prior to C# 8.0

case {

First: {Item: IComparable<T> first },

Second: {Item: T second } }:

if(first.CompareTo(second) < 0)

{

// first is less than second

}

else

{

// second is less than or equal to first

}

break;

default:

throw new InvalidCastException(

@$"Unable to sort the items. {

typeof(T) } does not support IComparable<T>");

}

\_SubItems = value;

}

}

private Pair<BinaryTree<T>?>? \_SubItems;

}

Unfortunately, if you now declare a BinaryTree<SomeType> class variable but the type argument (SomeType) does not implement the IComparable<SomeType> interface, there is no way to sort the items; in turn, we throw an InvalidCastException indicating the type doesn’t support the requisite interface. This eliminates a key reason for having generics in the first place: to improve type safety.

To avoid this exception and instead generate a compile-time error if the type argument does not implement the interface, C# allows you to supply an optional list of constraints for each type parameter declared in the generic type. A constraint declares the characteristics that the generic type requires of the type argument supplied for each type parameter. You declare a constraint using the where keyword, followed by a parameter–requirements pair, where the parameter must be one of those declared in the generic type. The requirements describe one of three things: the class or interfaces to which the type argument must be convertible, the presence of a default constructor, or a reference/value type restriction.

Interface Constraints

To ensure that a binary tree has its nodes correctly ordered, you can use the CompareTo() method in the BinaryTree class. To do this most effectively, you should impose a constraint on the T type parameter. That is, you need the T type parameter to implement the IComparable<T> interface. The syntax for declaring this constraint appears in Listing 12.22.

Listing 12.22: Declaring an Interface Constraint

public class BinaryTree<T>

where T: System.IComparable<T>

{

public BinaryTree(T item)

{

Item = item;

}

public T Item { get; set; }

public Pair<BinaryTree<T>> SubItems

{

get{ return \_SubItems; }

set

{

switch(value)

{

// Null handling removed for elucidation

// Using C# 8.0 Pattern Matching. Switch to

// checking for null prior to C# 8.0

case {

First: {Item: T first },

Second: {Item: T second } }:

if(first.CompareTo(second) < 0)

{

// first is less than second

}

else

{

// second is less than or equal to first

}

break;

default:

throw new InvalidCastException(

@$"Unable to sort the items. {

typeof(T) } does not support IComparable<T>");

}

\_SubItems = value;

}

}

private Pair<BinaryTree<T>?>? \_SubItems;

}

While the code change in this example is minor, it moves the error identification to the compiler rather than at runtime, and this is an important difference. When given the interface constraint addition in Listing 12.22, the compiler ensures that each time you use the BinaryTree<T> class, you specify a type parameter that implements the corresponding construction of the IComparable<T> interface. Furthermore, you no longer need to explicitly cast the variable to an IComparable<T> interface before calling the CompareTo() method. Casting is not even required to access members that use explicit interface implementation, which in other contexts would hide the member without a cast. When calling a method on a value typed as a generic type parameter, the compiler checks whether the method matches any method on any of the interfaces declared as constraints.

If you tried to create a BinaryTree<T> variable using System.Text.StringBuilder as the type parameter, you would receive a compiler error because StringBuilder does not implement IComparable<StringBuilder>. The error is similar to the one shown in Output 12.3.

|  |
| --- |
| Output 12.3 |
| error CS0311: The type 'System.Text.StringBuilder' cannot be used as type  parameter 'T' in the generic type or method 'BinaryTree<T>'. There is no  implicit reference conversion from 'System.Text.StringBuilder' to  'System.IComparable<System.Text.StringBuilder>'. |

To specify an interface for the constraint, you declare an interface type constraint. This constraint even circumvents the need to cast to call an explicit interface member implementation.

Type Parameter Constraints

Sometimes you might want to constrain a type argument to be convertible to a particular type. You do this using a type parameter constraint, as shown in Listing 12.23.

Listing 12.23: Declaring a Class Type Constraint

public class EntityDictionary<TKey, TValue>

: System.Collections.Generic.Dictionary<TKey, TValue>

where TKey: notnull

where TValue : EntityBase

{

...

}

In Listing 12.23, EntityDictionary<TKey, TValue> requires that all type arguments provided for the type parameter TValue be implicitly convertible to the EntityBase class. By requiring the conversion, it becomes possible to use the members of EntityBase on values of type TValue within the generic implementation, because the constraint will ensure that all type arguments can be implicitly converted to the EntityBase class.

The syntax for the class type constraint is the same as that for the interface type constraint, except that class type constraints must appear before any interface type constraints (just as the base class must appear before implemented interfaces in a class declaration). However, unlike interface constraints, multiple base class constraints are not allowed, since it is not possible to derive from multiple unrelated classes. Similarly, base class constraints cannot specify sealed classes or non-class types. For example, C# does not allow a type parameter to be constrained to string or System.Nullable<T> because there would then be only one possible type argument for that type parameter—that’s hardly “generic.” If the type parameter is constrained to a single type, there is no need for the type parameter in the first place; just use that type directly.

Certain “special” types are not legal as class type constraints. See Advanced Topic: Constraint Limitations, later in this chapter, for details.

\*\*\*COMP: Insert “Begin 7.3” tab

Starting with C# 7.3, you can use System.Enum as a constraint, thereby ensuring a type parameter is an enum. However, you cannot specify type System.Array as a constraint. The latter restriction has minimal impact, however, as other collection types and interfaces are preferable anyway—see Chapter 15.

|  |  |
| --- | --- |
|  | |
|  | AdVanced Topic  Delegate Constraints  C# 7.3 also added support for using System.Delegate (and System.MulticastDelegate). This allows for combining (using the static Combine() method) and separating (using the static Remove() method) delegates in a type-safe manner. There isn’t a strongly typed way for a generic type to invoke a delegate, but the DynamicInvoke() method can accomplish this. Internally, it uses reflection. Even though the generic type can’t invoke the delegate directly (without going through DynamicInvoke()), it is possible via a direct reference to T at compile time. For example, you could invoke a Combine() method and cast to the expected type as shown with the pattern matching in Listing 12.24.  Listing 12.24: Declaring a Generic with a MulticastDelegate Constraint  static public object? InvokeAll<TDelegate>(  object?[]? args, params TDelegate[] delegates)  // Constraint of type Action/Func not allowed  where TDelegate : System.MulticastDelegate  {  switch (Delegate.Combine(delegates))  {  case Action action:  action();  return null;  case TDelegate result:  return result.DynamicInvoke(args);  default:  return null;  };  }  In this example, we attempt to cast to Action before invoking the result. If that effort is not successful, we cast to TDelegate and invoke the result using DynamicInvoke().  Note that outside the generic we would know the type of T, so we could invoke it directly after calling Combine():  Action? result =   (Action?)Delegate.Combine(actions);  result?.Invoke();  Note the comment in Listing 12.24. While System.Delegate and System.MulticastDelegate are supported, you cannot specify a specific delegate type such as Action, Func<T>, or one of the related types. |
|  | |

unmanaged Constraint

C# 7.3 introduced the unmanaged constraint, which limits the type parameter to be of type sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal, bool, an enum, a pointer, or any struct where all fields are unmanaged. This allows you to do things like use the sizeof (or stackalloc, as discussed in Chapter 22) operator on an unmanaged constrained type parameter.

\*\*\*COMP: Insert “Begin 8.0” tab

Prior to C# 8.0, the unmanaged constraint restricted type parameters to constructed struct types—that is, value types that were not generic. However, C# 8.0 removed this constraint. You can now declare a variable of type Thing<Thing<int>> even if Thing<T> had an unmanaged constraint for T.

\*\*\*COMP: Insert “End 7.3” tab

notnull Constraint

In Listing 12.23, there is a second constraint: the non-null constraint using the contextual keyword notnull. This constraint triggers a warning if a nullable type is specified for the type parameter decorated with notnull. In this case, for example, declaring EntityDictionary<string?, EntityBase> will lead to the following warning: Nullability of type argument 'string?' doesn't match 'notnull' constraint.

The notnull keyword cannot be combined with the struct or class constraints, which are not nullable by default (as we describe next).

struct/class Constraints

Another valuable generic constraint is the ability to restrict type arguments to be any non-nullable value type or any reference type. Rather than specifying a class from which T must derive, you simply use the keyword struct or class, as shown in Listing 12.25.

Listing 12.25: Specifying the Type Parameter as a Value Type

public struct Nullable<T> :

IFormattable, IComparable,

IComparable<Nullable<T>>, INullable

where T : struct

{

// ...

public static implicit operator T?(T value) =>

new T?(value);

public static explicit operator T(T? value) => value!.Value;

}

Note that the class constraint restricts the type parameter to reference types including interface, delegate, or array types (and not, as the keyword might seem to imply, only class types).

In C# 8.0, a class constraint defaults to not nullable (assuming, of course, that nullable reference types are enabled). Specifying a nullable reference type parameter will trigger a warning by the compiler. You can change the generic type to allow nullable reference types by including the nullable modifier on the class constraint. Consider the WeakReference<T> class introduced in Chapter 10. Since only reference types are garbage collected, this type includes the class constraint as follows:

public sealed partial class WeakReference<T> : ISerializable

where T : class?

{ ... }

This restricts the type parameter, T, to be a reference type—which may be nullable.

In contrast to the class constraint, the struct constraint does not allow the nullable modifier. Instead, you can specify nullability when using the parameter. In Listing 12.25, for example, the implicit and explicit conversion operators, for example, use T and T? to identify whether a non-nullable or nullable version of T is allowed. As such, the type parameter is constrained when it is used in member declarations, rather than with a type constraint.

Because a class type constraint requires a reference type, using a struct constraint with a class type constraint would be contradictory. Therefore, you cannot combine struct and class constraints.

The struct constraint has one special characteristic: Nullable value types do not satisfy the constraint. Why? Nullable value types are implemented as the generic type Nullable<T>, which itself applies the struct constraint to T. If nullable value types satisfied that constraint, it would be possible to define the nonsense type Nullable<Nullable<int>>. A doubly nullable integer is confusing to the point of being meaningless. (As expected, the shorthand syntax int?? is also disallowed.)

\*\*\*COMP: Insert “End 8.0” tab

Multiple Constraints

For any given type parameter, you may specify any number of interface type constraints, but no more than one class type constraint (just as a class may implement any number of interfaces but inherit from only one other class). Each new constraint is declared in a comma-delimited list following the generic type parameter and a colon. If there is more than one type parameter, each must be preceded by the where keyword. In Listing 12.26, the generic EntityDictionary class declares two type parameters: TKey and TValue. The TKey type parameter has two interface type constraints, and the TValue type parameter has one class type constraint.

Listing 12.26: Specifying Multiple Constraints

public class EntityDictionary<TKey, TValue>

: Dictionary<TKey, TValue>

where TKey : IComparable<TKey>, IFormattable

where TValue : EntityBase

{

...

}

In this case, there are multiple constraints on TKey itself and an additional constraint on TValue. When specifying multiple constraints on one type parameter, an AND relationship is assumed. If a type C is supplied as the type argument for TKey, C must implement IComparable<C> and IFormattable, for example.

Notice there is no comma between each where clause.

Constructor Constraints

In some cases, it is desirable to create an instance of the type argument’s type inside the generic class. In Listing 12.27, for example, the MakeValue() method for the EntityDictionary<TKey, TValue> class must create an instance of the type argument corresponding to type parameter TValue.

Listing 12.27: Requiring a Default Constructor Constraint

public class EntityBase<TKey>

where TKey: notnull

{

public EntityBase(TKey key)

{

Key = key;

}

public TKey Key { get; set; }

}

public class EntityDictionary<TKey, TValue> :

Dictionary<TKey, TValue>

where TKey: IComparable<TKey>, IFormattable

where TValue : EntityBase<TKey>, new()

{

// ...

public TValue MakeValue(TKey key)

{

TValue newEntity = new TValue

{

Key = key;

}

Add(newEntity.Key, newEntity);

return newEntity;

}

// ...

}

Because not all objects are guaranteed to have public default constructors, the compiler does not allow you to call the default constructor on an unconstrained type parameter. To override this compiler restriction, you can add the text new() after all other constraints are specified. This text, called a constructor constraint, requires the type argument corresponding to the constrained type parameter to have a public or internal default constructor. Only the default constructor constraint is available. You cannot specify a constraint that ensures that the type argument supplied provides a constructor that takes formal parameters.

Listing 12.27 includes a constructor constraint that forces the type argument supplied for TValue to provide a public parameterless constructor. There is no constraint to force the type argument to provide a constructor that takes other formal parameters. For example, you might want to constrain TValue so that the type argument provided for it must provide a constructor that takes the type argument provided for TKey, but this is not possible. Listing 12.28 demonstrates the invalid code.

Listing 12.28: Constructor Constraints Can Be Specified Only for Default Constructors

public TValue New(TKey key)

{

// Error: 'TValue': Cannot provide arguments

// when creating an instance of a variable type

TValue newEntity = null;

// newEntity = new TValue(key);

Add(newEntity.Key, newEntity);

return newEntity;

}

One way to circumvent this restriction is to supply a factory interface that includes a method for instantiating the type. The factory implementing the interface takes responsibility for instantiating the entity rather than the EntityDictionary itself (see Listing 12.29).

Listing 12.29: Using a Factory Interface in Place of a Constructor Constraint

public class EntityBase<TKey>

{

public EntityBase(TKey key)

{

Key = key;

}

public TKey Key { get; set; }

}

public class EntityDictionary<TKey, TValue, TFactory> :

Dictionary<TKey, TValue>

where TKey : IComparable<TKey>, IFormattable

where TValue : EntityBase<TKey>

where TFactory : IEntityFactory<TKey, TValue>, new()

{

...

public TValue New(TKey key)

{

TFactory factory = new TFactory();

TValue newEntity = factory.CreateNew(key);

Add(newEntity.Key, newEntity);

return newEntity;

}

...

}

public interface IEntityFactory<TKey, TValue>

{

TValue CreateNew(TKey key);

}

...

A declaration such as this allows you to pass the new key to a TValue factory method that takes parameters, rather than forcing you to rely on the default constructor. It no longer uses the constructor constraint on TValue because TFactory is responsible for instantiating value. (One modification to the code in Listing 12.29 would be to cache a reference to the factory method—possibly leveraging Lazy<T> if multithreaded support was needed. This would enable you to reuse the factory method instead of reinstantiating it every time.)

A declaration for a variable of type EntityDictionary<TKey, TValue, TFactory> would result in an entity declaration similar to the Order entity in Listing 12.30.

Listing 12.30: Declaring an Entity to Be Used in EntityDictionary<...>

public class Order : EntityBase<Guid>

{

public Order(Guid key) :

base(key)

{

// ...

}

}

public class OrderFactory : IEntityFactory<Guid, Order>

{

public Order CreateNew(Guid key)

{

return new Order(key);

}

}

Constraint Inheritance

Neither generic type parameters nor their constraints are inherited by a derived class, because generic type parameters are not members. (Remember, class inheritance is the property that the derived class has all the members of the base class.) It is a common practice to make new generic types that inherit from other generic types. In such a case, because the type parameters of the derived generic type become the type arguments of the generic base class, the type parameters must have constraints equal to (or stronger than) those on the base class. Confused? Consider Listing 12.31.

Listing 12.31: Inherited Constraints Specified Explicitly

class EntityBase<T> where T : IComparable<T>

{

// ...

}

// ERROR:

// The type 'U' must be convertible to

// 'System.IComparable<U>' to use it as parameter

// 'T' in the generic type or method

// class Entity<U> : EntityBase<U>

// {

// ...

// }

In Listing 12.31, EntityBase<T> requires that the type argument U supplied for T by the base class specifier EntityBase<U> implement IComparable<U>. Therefore, the Entity<U> class needs to require the same constraint on U. Failure to do so will result in a compile-time error. This pattern increases a programmer’s awareness of the base class’s type constraint in the derived class, avoiding the confusion that might otherwise occur if the programmer uses the derived class and discovers the constraint but does not understand where it comes from.

We have not covered generic methods yet; we’ll get to them later in this chapter. For now, simply recognize that methods may also be generic and may also place constraints on the type arguments supplied for their type parameters. How, then, are constraints handled when a virtual generic method is inherited and overridden? In contrast to the situation with type parameters declared on a generic class, constraints on overriding virtual generic methods (or explicit interface) methods are inherited implicitly and may not be restated (see Listing 12.32).

Listing 12.32: Repeating Inherited Constraints on Virtual Members Is Prohibited

class EntityBase

{

public virtual void Method<T>(T t)

where T : IComparable<T>

{

// ...

}

}

class Order : EntityBase

{

public override void Method<T>(T t)

// Constraints may not be repeated on overriding

// members

// where T : IComparable<T>

{

// ...

}

}

In the generic class inheritance case, the type parameter on the derived class can be further constrained by adding not only the constraints on the base class (required), but also other constraints. However, overriding virtual generic methods need to conform exactly to the constraints defined by the base class method. Additional constraints could break polymorphism, so they are not allowed and the type parameter constraints on the overriding method are implied.

|  |  |
| --- | --- |
|  | |
|  | AdVanced Topic  Constraint Limitations  Constraints are appropriately limited to avoid nonsensical code. For example, you cannot combine a class type constraint with a struct or class constraint, or notnull with either type of constraint. Also, you cannot specify constraints that restrict inheritance to special types such as object, arrays, or System.ValueType. As mentioned earlier, System.Enum (enum), System.Delegate, and System.MulticastDelegate are supported starting in C# 7.3. However, you cannot have a constraint for a specific delegate type like Action, Func<T>, or related types.  In some cases, constraint limitations are perhaps more desirable, yet they are still not supported. Only having the ability to require a default constructor is perhaps one such limitation. The following subsections provide some additional examples of constraints that are not allowed.  Operator Constraints Are Not Allowed  All generics implicitly allow for == and != comparisons, along with implicit casts to object, since everything is an object. You cannot constrain a type parameter to a type that implements a particular method or operator (other than in the aforementioned cases), except via interface type constraints (for methods) or class type constraints (for methods and operators). Because of this, the generic Add() in Listing 12.33 does not work.  Listing 12.33: Constraint Expressions Cannot Require Operators  public abstract class MathEx<T>  {  public static T Add(T first, T second)  {  // Error: Operator '+' cannot be applied to  // operands of type 'T' and 'T'  // return first + second;  }  }  In this case, the method assumes that the + operator is available on all types that could be supplied as type arguments for T. But there is no constraint that prevents you from supplying a type argument that does not have an associated addition operator, so an error occurs. Unfortunately, there is no way to specify that an addition operator is required within a constraint, aside from using a class type constraint where the class type implements an addition operator.  More generally, there is no way to constrain a type to have a static method.  OR Criteria Are Not Supported  If you supply multiple interfaces or class constraints for a type parameter, the compiler always assumes an AND relationship between constraints. For example, where T : IComparable<T>, IFormattable requires that both IComparable<T> and IFormattable are supported. There is no way to specify an OR relationship between constraints. Hence, code equivalent to Listing 12.34 is not supported.  Listing 12.34: Combining Constraints Using an OR Relationship Is Not Allowed  public class BinaryTree<T>  // Error: OR is not supported  // where T: System.IComparable<T> || System.IFormattable  {  ...  }  Supporting this functionality would prevent the compiler from resolving which method to call at compile time. |
|  | |

\*\*\*COMP: End Advanced Topic after Listing 12.34

Generic Methods

Earlier, you saw that it is a relatively simple matter to add a method to a type when the type is generic; such a method can use the generic type parameters declared by the type. You did this, for example, in the generic class examples we have seen so far.

Generic methods use generic type parameters, much as generic types do. They can be declared in generic or nongeneric types. If declared in a generic type, their type parameters are distinct from those of their containing generic type. To declare a generic method, you specify the generic type parameters the same way you do for generic types: Add the type parameter declaration syntax immediately following the method name, as shown in the MathEx.Max<T> and MathEx.Min<T> examples in Listing 12.35.

Listing 12.35: Defining Generic Methods

public static class MathEx

{

public static T Max<T>(T first, params T[] values)

where T : IComparable<T>

{

T maximum = first;

foreach (T item in values)

{

if (item.CompareTo(maximum) > 0)

{

maximum = item;

}

}

return maximum;

}

public static T Min<T>(T first, params T[] values)

where T : IComparable<T>

{

T minimum = first;

foreach (T item in values)

{

if (item.CompareTo(minimum) < 0)

{

minimum = item;

}

}

return minimum;

}

}

In this example, the method is static, although this is not required.

Generic methods, like generic types, can include more than one type parameter. The arity (the number of type parameters) is an additional distinguishing characteristic of a method signature. That is, it is legal to have two methods with identical names and formal parameter types, as long as they differ in method type parameter arity.

Generic Method Type Inference

Just as type arguments are provided after the type name when using a generic type, so the method type arguments are provided after the method type name. The code used to call the Min<T> and Max<T> methods looks like that shown in Listing 12.36.

Listing 12.36: Specifying the Type Parameter Explicitly

Console.WriteLine(

MathEx.Max<int>(7, 490));

Console.WriteLine(

MathEx.Min<string>("R.O.U.S.", "Fireswamp"));

The results of Listing 12.36 appear in Output 12.4.

|  |
| --- |
| Output 12.4 |
| 490  Fireswamp |

Not surprisingly, the type arguments, int and string, correspond to the actual types used in the generic method calls. However, specifying the type arguments is redundant because the compiler can infer the type parameters from the formal parameters passed to the method. Clearly, the caller of Max in Listing 12.36 intends the type argument to be int because both of the method arguments are of type int. To avoid redundancy, you can exclude the type parameters from the call in all cases when the compiler is able to logically infer which type arguments you must have intended. An example of this practice, which is known as method type inference, appears in Listing 12.37. The results appear in Output 12.5.

Listing 12.37: Inferring the Type Argument from the Arguments

Console.WriteLine(

MathEx.Max(7, 490)); // No type arguments!

Console.WriteLine(

MathEx.Min("R.O.U.S'", "Fireswamp"));

|  |
| --- |
| Output 12.5 |
| 490  Fireswamp |

For method type inference to succeed, the types of the arguments must be “matched” with the formal parameters of the generic method in such a way that the desired type arguments can be inferred. An interesting question to consider is what happens when contradictory inferences are made. For example, when you call the Max<T> method using MathEx.Max(7.0, 490), the compiler could deduce from the first argument that the type argument should be double, and it could deduce from the second argument that the type argument is int—a contradiction. In C# 2.0, this discrepancy would have produced an error. A more sophisticated analysis would notice that the contradiction can be resolved because every int can be converted to a double, so double is the best choice for the type argument. C# 3.0 and C# 4.0 both included improvements to the method type inferencing algorithm that permit the compiler to make these more sophisticated analyses.

In cases where method type inference is still not sophisticated enough to deduce the type arguments, you can resolve the error either by inserting casts on the arguments that clarify to the compiler the argument types that should be used in the inferences or by giving up on type inferencing and including the type arguments explicitly.

Notice that the method type inference algorithm, when making its inferences, considers only the arguments, the arguments’ types, and the formal parameter types of the generic method. Other factors that could, in practice, be used in the analysis—such as the return type of the generic method, the type of the variable to which the method’s returned value is being assigned, or the constraints on the method’s generic type parameters—are not considered at all by this algorithm.

Specifying Constraints

Type parameters of generic methods may be constrained in the same way that type parameters of generic types are constrained. For example, you can restrict a method’s type parameter to implement an interface or to be convertible to a class type. The constraints are specified between the argument list and the method body, as shown in Listing 12.38.

Listing 12.38: Specifying Constraints on Generic Methods

public class ConsoleTreeControl

{

// Generic method Show<T>

public static void Show<T>(BinaryTree<T> tree, int indent)

where T : IComparable<T>

{

Console.WriteLine("\n{0}{1}",

"+ --".PadLeft(5\*indent, ' '),

tree.Item.ToString());

if (tree.SubItems.First != null)

Show(tree.SubItems.First, indent+1);

if (tree.SubItems.Second != null)

Show(tree.SubItems.Second, indent+1);

}

}

Here, the Show<T> implementation itself does not directly use any member of the IComparable<T> interface, so you might wonder why the constraint is required. Recall, however, that the BinaryTree<T> class did require this constraint (see Listing 12.39).

Listing 12.39: BinaryTree<T> Requiring IComparable<T> Type Parameters

public class BinaryTree<T>

where T: System.IComparable<T>

{

...

}

Because the BinaryTree<T> class requires this constraint on its T, and because Show<T> uses its T as a type argument corresponding to a constrained type parameter, Show<T> needs to ensure that the constraint on the class’s type parameter is met on its method type argument.

|  |  |
| --- | --- |
|  | |
|  | AdVanced Topic  Casting inside a Generic Method  Sometimes you should be wary of using generics—for instance, when using them specifically to bury a cast operation. Consider the following method, which converts a stream into an object of a given type:  public static T Deserialize<T>(  Stream stream, IFormatter formatter)  {  return (T)formatter.Deserialize(stream);  }  The formatter is responsible for removing data from the stream and converting it to an object. The Deserialize() call on the formatter returns data of type object. A call to use the generic version of Deserialize() looks something like this:  string greeting =  Deserialization.Deserialize<string>(stream, formatter);  The problem with this code is that to the caller of the method, Deserialize<T>() appears to be type-safe. However, a cast operation is still performed on behalf of the caller, as in the case of the nongeneric equivalent shown here:  string greeting =  (string)Deserialization.Deserialize(stream, formatter);  The cast could fail at runtime; the method might not be as type-safe as it appears. The Deserialize<T> method is generic solely so that it can hide the existence of the cast from the caller, which seems dangerously deceptive. It might be better for the method to be nongeneric and return object, making the caller aware that it is not type-safe. Developers should use care when casting in generic methods if there are no constraints to verify cast validity.   |  | | --- | | Guidelines  AVOID misleading the caller with generic methods that are not as type-safe as they appear. | |
|  | |

\*\*\*COMP: End Advanced Topic after Guidelines

Covariance and Contravariance

A question often asked by new users of generic types is why an expression of type List<string> may not be assigned to a variable of type List<object>: If a string may be converted to type object, surely a list of strings is similarly compatible with a list of objects. In reality, this is not, generally speaking, either type-safe or legal. If you declare two variables with different type parameters using the same generic class, the variables are not type-compatible even if they are assigning from a more specific type to a more generic type—in other words, they are not covariant.

Covariant is a technical term from category theory, but its underlying idea is straightforward: Suppose two types X and Y have a special relationship—namely, that every value of the type X may be converted to the type Y. If the types I<X> and I<Y> always also have that same special relationship, we say, “I<T> is covariant in T.” When dealing with simple generic types with only one type parameter, the type parameter can be understood such that we simply say, “I<T> is covariant.” The conversion from I<X> to I<Y> is called a covariant conversion.

For example, two instances of a generic class, Pair<Contact> and Pair<PdaItem>, are not type-compatible even when the type arguments are themselves compatible. In other words, the compiler prevents the conversion (implicit or explicit) of Pair<Contact> to Pair<PdaItem>, even though Contact derives from PdaItem. Similarly, converting Pair<Contact> to the interface type IPair<PdaItem> will fail. See Listing 12.40 for an example.

Listing 12.40: Conversion between Generics with Different Type Parameters

// ...

// Error: Cannot convert type ...

Pair<PdaItem> pair = (Pair<PdaItem>) new Pair<Contact>();

IPair<PdaItem> duple = (IPair<PdaItem>) new Pair<Contact>();

But why is this not legal? Why are List<T> and Pair<T> not covariant? Listing 12.41 shows what would happen if the C# language allowed unrestricted generic covariance.

Listing 12.41: Preventing Covariance Maintains Homogeneity

//...

Contact contact1 = new Contact("Princess Buttercup"),

Contact contact2 = new Contact("Inigo Montoya");

Pair<Contact> contacts = new Pair<Contact>(contact1, contact2);

// This gives an error: Cannot convert type ...,

// but suppose it did not

// IPair<PdaItem> pdaPair = (IPair<PdaItem>) contacts;

// This is perfectly legal but not type-safe

// pdaPair.First = new Address("123 Sesame Street");

...

An IPair<PdaItem> can contain an address, but the object is really a Pair<Contact> that can contain only contacts, not addresses. Type safety is completely violated if unrestricted generic covariance is allowed.

Now it should also be clear why a list of strings may not be used as a list of objects. You cannot insert an integer into a list of strings, but you can insert an integer into a list of objects; thus it must be illegal to cast a list of strings to a list of objects—an error the compiler can enforce.

\*\*\*COMP: Insert “Begin 4.0” tab

Enabling Covariance with the out Type Parameter Modifier in C# 4.0 (and Later)

You might have noticed that both problems described earlier as consequences of unrestricted covariance arise because the generic pair and the generic list allow their contents to be written. Suppose we eliminated this possibility by creating a read-only IReadOnlyPair<T> interface that exposes T only as coming “out” of the interface (i.e., used as the return type of a method or read-only property) and never going “into” it (i.e., used as a formal parameter or writeable property type). If we restricted ourselves to an “out-only” interface with respect to T, the covariance problem just described would not occur (see Listing 12.42).

Listing 12.42: Potentially Possible Covariance

interface IReadOnlyPair<T>

{

T First { get; }

T Second { get; }

}

interface IPair<T>

{

T First { get; set; }

T Second { get; set; }

}

public struct Pair<T> : IPair<T>, IReadOnlyPair<T>

{

// ...

}

class Program

{

static void Main()

{

// Error: Only theoretically possible without

// the out type parameter modifier

Pair<Contact> contacts =

new Pair<Contact>(

new Contact("Princess Buttercup"),

new Contact("Inigo Montoya") );

IReadOnlyPair<PdaItem> pair = contacts;

PdaItem pdaItem1 = pair.First;

PdaItem pdaItem2 = pair.Second;

}

}

When we restrict the generic type declaration to expose data only as it comes out of the interface, there is no reason for the compiler to prevent covariance. All operations on an IReadOnlyPair<PdaItem> instance would convert Contacts (from the original Pair<Contact> object) up to the base class PdaItem—a perfectly valid conversion. There is no way to “write” an address into the object that is really a pair of contacts, because the interface does not expose any writeable properties.

The code in Listing 12.42 still does not compile. However, support for safe covariance was added to C# 4. To indicate that a generic interface is intended to be covariant in one of its type parameters, you can declare the type parameter with the out type parameter modifier. Listing 12.43 shows how to modify the interface declaration to indicate that it should be allowed to be covariant.

Listing 12.43: Covariance Using the out Type Parameter Modifier

...

interface IReadOnlyPair<out T>

{

T First { get; }

T Second { get; }

}

Modifying the type parameter on the IReadOnlyPair<out T> interface with out will cause the compiler to verify that T is, indeed, used only for “outputs”—method return types and read-only property return types—and never for formal parameters or property setters. From then on, the compiler will allow any covariant conversions involving the interface to succeed. When this modification is made to the code in Listing 12.42, it will compile and execute successfully.

Several important restrictions are placed on covariant conversions:

* Only generic interfaces and generic delegates (described in Chapter 13) may be covariant. Generic classes and structs are never covariant.

The varying type arguments of both the source and target generic types must be reference types, not value types. That is, an IReadOnlyPair<string> may be converted covariantly to IReadOnlyPair<object> because both string and IReadOnlyPair<object> are reference types. An IReadOnlyPair<int> may not be converted to IReadOnlyPair<object> because int is not a reference type.

The interface or delegate must be declared as supporting covariance, and the compiler must be able to verify that the annotated type parameters are, in fact, used in only “output” positions.

Enabling Contravariance with the in Type Parameter Modifier in C# 4.0 (and Later)

Covariance that “goes backward” is called contravariance. Again, suppose two types X and Y are related such that every value of the type X may be converted to the type Y. If the types I<X> and I<Y> always have that same special relationship “backward”—that is, every value of the type I<Y> can be converted to the type I<X>—we say that “I<T> is contravariant in T.”

Most people find that contravariance is much harder to comprehend than covariance is. The canonical example of contravariance is a comparer. Suppose you have a derived type, Apple, and a base type, Fruit. Clearly, they have the special relationship: Every value of type Apple may be converted to Fruit.

Now suppose you have an interface ICompareThings<T> that has a method bool FirstIsBetter(T t1, T t2) that takes two Ts and returns a bool saying whether the first one is better than the second one.

What happens when we provide type arguments? An ICompareThings<Apple> has a method that takes two Apples and compares them. An ICompareThings<Fruit> has a method that takes two Fruits and compares them. But since every Apple is a Fruit, clearly a value of type ICompareThings<Fruit> can be safely used anywhere that an ICompareThings<Apple> is needed. The direction of the convertibility has been reversed—hence the term contravariance.

Perhaps unsurprisingly, the opposite restrictions to those placed on a covariant interface are necessary to ensure safe contravariance. An interface that is contravariant in one of its type parameters must use that type parameter only in input positions such as formal parameters (or in the types of write-only properties, which are extremely rare). You can mark an interface as being contravariant by declaring the type parameter with the in modifier, as shown in Listing 12.44.

Listing 12.44: Contravariance Using the in Type Parameter Modifier

class Fruit {}

class Apple : Fruit {}

class Orange : Fruit {}

interface ICompareThings<in T>

{

bool FirstIsBetter(T t1, T t2);

}

class Program

{

class FruitComparer : ICompareThings<Fruit>

{ ... }

static void Main()

{

// Allowed in C# 4.0 and later

ICompareThings<Fruit> fc = new FruitComparer();

Apple apple1 = new Apple();

Apple apple2 = new Apple();

Orange orange = new Orange();

// A fruit comparer can compare apples and oranges:

bool b1 = fc.FirstIsBetter(apple1, orange);

// or apples and apples:

bool b2 = fc.FirstIsBetter(apple1, apple2);

// This is legal because the interface is

// contravariant

ICompareThings<Apple> ac = fc;

// This is really a fruit comparer, so it can

// still compare two apples

bool b3 = ac.FirstIsBetter(apple1, apple2);

}

}

Like covariance support, contravariance uses a type parameter modifier: in, which appears in the interface’s type parameter declaration. This instructs the compiler to check that T never appears on a property getter or as the return type of a method, thereby enabling contravariant conversions for this interface.

Contravariant conversions have all the analogous restrictions as described earlier for covariant conversions: They are valid only for generic interface and delegate types, the varying type arguments must be reference types, and the compiler must be able to verify that the interface is safe for the contravariant conversions.

An interface can be covariant in one type parameter and contravariant in another, although this case seldom arises in practice except with delegates. The Func<A1, A2, ..., R> family of delegates, for example, are covariant in the return type, R, and contravariant in all the argument types.

Lastly, note that the compiler will check the validity of the covariance and contravariance type parameter modifiers throughout the source. Consider the PairInitializer<in T> interface in Listing 12.45.

Listing 12.45: Compiler Validation of Variance

// ERROR: Invalid variance; the type parameter 'T' is not

// invariantly valid

interface IPairInitializer<in T>

{

void Initialize(IPair<T> pair);

}

// Suppose the code above were legal, and see what goes

// wrong:

class FruitPairInitializer : IPairInitializer<Fruit>

{

// Let’s initialize our pair of fruits with an

// apple and an orange:

public void Initialize(IPair<Fruit> pair)

{

pair.First = new Orange();

pair.Second = new Apple();

}

}

// ... later ...

var f = new FruitPairInitializer();

// This would be legal if contravariance were legal:

IPairInitializer<Apple> a = f;

// And now we write an orange into a pair of apples:

a.Initialize(new Pair<Apple>());

A casual observer might be tempted to think that since IPair<T> is used only as an input formal parameter, the contravariant in modifier on IPairInitializer is valid. However, the IPair<T> interface cannot safely vary, so it cannot be constructed with a type argument that can vary. As you can see, this would not be type-safe and, in turn, the compiler disallows the IPairInitializer<T> interface from being declared as contravariant in the first place.

Support for Unsafe Covariance in Arrays

So far, we have described covariance and contravariance as being properties of generic types. Of all the nongeneric types, arrays are most like generics; that is, just as we think of a generic “list of T” or a generic “pair of T,” so we can think of an “array of T” as demonstrating the same sort of pattern. Since arrays clearly support both reading and writing, given what you know about covariance and contravariance, you probably would suppose that arrays may be neither safely contravariant nor covariant. That is, you might imagine that an array can be safely covariant only if it is never written to and safely contravariant only if it is never read from—though neither seems like a realistic restriction.

Unfortunately, C# does support array covariance, even though doing so is not type-safe. For example, Fruit[] fruits = new Apple[10]; is perfectly legal in C#. If you then include the expression fruits[0] = new Orange();, the runtime will issue a type-safety violation in the form of an exception. It is deeply disturbing that it is not always legal to assign an Orange into an array of Fruit because it might really be an array of Apples, but that is the situation not just in C#, but in all CLR languages that use the runtime’s implementation of arrays.

You should try to avoid using unsafe array covariance. Every array is convertible to the read-only (and therefore safely covariant) interface IEnumerable<T>; that is, IEnumerable<Fruit> fruits = new Apple[10] is both safe and legal because there is no way to insert an Orange into the array if all you have is the read-only interface.

|  |
| --- |
| Guidelines  AVOID unsafe array covariance. Instead, CONSIDER converting the array to the read-only interface IEnumerable<T>, which can be safely converted via covariant conversions. |

\*\*\*COMP: Insert “End 4.0” tab

Generic Internals

Given the discussions in earlier chapters about the prevalence of objects within the CLI type system, it should come as no surprise to learn that generics are also objects. In fact, the type parameter on a generic class becomes metadata that the runtime uses to build appropriate classes when needed. Generics, therefore, support inheritance, polymorphism, and encapsulation. With generics, you can define methods, properties, fields, classes, interfaces, and delegates.

To achieve this, generics require support from the underlying runtime. In turn, the addition of generics to the C# language is a feature of both the compiler and the framework. To avoid boxing, for example, the implementation of generics is different for value-based type parameters than for generics with reference type parameters.

|  |  |
| --- | --- |
|  | |
|  | AdVanced Topic  CIL Representation of Generics  When a generic class is compiled, it is not significantly different from a nongeneric class. The result of the compilation consists of just metadata and CIL. The CIL is parameterized to accept a user-supplied type somewhere in code. As an example, suppose you had a simple Stack class declared as shown in Listing 12.46.  Listing 12.46: Stack<T> Declaration  public class Stack<T> where T : IComparable  {  private T[] \_Items;  // rest of the class here  }  When you compile the class, the generated CIL is parameterized and looks something like Listing 12.47.  Listing 12.47: CIL Code for Stack<T>  .class private auto ansi beforefieldinit  Stack'1<([mscorlib]System.IComparable)T>  extends [mscorlib]System.Object  {  ...  }  The first notable item is the '1 that appears following Stack on the second line. That number is the arity of the generic types: It declares the number of type parameters for which the generic class will require type arguments. A declaration such as EntityDictionary<TKey, TValue> would have an arity of 2.  The second line of the generated CIL shows the constraints imposed upon the class. The T type parameter is decorated with an interface declaration for the IComparable constraint.  If you continue looking through the CIL, you will find that the item’s array declaration of type T is altered to contain a type parameter using exclamation point notation, which is featured in the generics-capable version of the CIL. The exclamation point denotes the presence of the first type parameter specified for the class, as shown in Listing 12.48.  Listing 12.48: CIL with Exclamation Point Notation to Support Generics  .class public auto ansi beforefieldinit  'Stack'1'<([mscorlib]System.IComparable) T>  extends [mscorlib]System.Object  {  .field private !0[ ] \_Items  ...  }  Beyond the inclusion of the arity and type parameter in the class header and the type parameter denoted with exclamation points in code, there is little difference between the CIL generated for a generic class and the CIL generated for a nongeneric class. |
|  | |

|  |  |
| --- | --- |
|  | |
|  | AdVanced Topic  Instantiating Generics Based on Value Types  When a generic type is first constructed with a value type as a type parameter, the runtime creates a specialized generic type with the supplied type parameter(s) placed appropriately in the CIL. Therefore, the runtime creates new specialized generic types for each new parameter value type.  For example, suppose some code declared a Stack constructed of integers, as shown in Listing 12.49.  Listing 12.49: Stack<int> Definition  Stack<int> stack;  When using this type, Stack<int>, for the first time, the runtime generates a specialized version of the Stack class with the type argument int substituted for its type parameter. From then on, whenever the code uses a Stack<int>, the runtime reuses the generated specialized Stack<int> class. In Listing 12.50, you declare two instances of a Stack<int>, both using the code already generated by the runtime for a Stack<int>.  Listing 12.50: Declaring Variables of Type Stack<T>  Stack<int> stackOne = new Stack<int>();  Stack<int> stackTwo = new Stack<int>();  If, later in the code, you create another Stack with a different value type substituted for the type parameter (such as a long or a user-defined struct), the runtime will generate another version of the generic type. The benefit of specialized value type classes is better performance. Furthermore, the code can avoid conversions and boxing because each specialized generic class natively contains the value type. |
|  | |

|  |  |
| --- | --- |
|  | |
|  | AdVanced Topic  Instantiating Generics Based on Reference Types  Generics work slightly differently for reference types. The first time a generic type is constructed with a reference type, the runtime creates a specialized generic type with object references substituted for type parameters in the CIL, rather than a specialized generic type based on the type argument. Each subsequent time a constructed type is instantiated with a reference type parameter, the runtime reuses the previously generated version of the generic type, even if the reference type is different from the first reference type.  For example, suppose you have two reference types: a Customer class and an Order class. Next, you create an EntityDictionary of Customer types:  EntityDictionary<Guid, Customer> customers;  Prior to accessing this class, the runtime generates a specialized version of the EntityDictionary class that, instead of storing Customer as the specified data type, stores object references. Suppose the next line of code creates an EntityDictionary of another reference type, called Order:  EntityDictionary<Guid, Order> orders =  new EntityDictionary<Guid, Order>();  Unlike with value types, no new specialized version of the EntityDictionary class is created for the EntityDictionary that uses the Order type. Instead, an instance of the version of EntityDictionary that uses object references is instantiated and the orders variable is set to reference it.  To still gain the advantage of type safety, for each object reference substituted in place of the type parameter, an area of memory for an Order type is specifically allocated and the pointer is set to that memory reference. Suppose you then encountered a line of code to instantiate an EntityDictionary of a Customer type as follows:  customers = new EntityDictionary<Guid, Customer>();  As with the previous use of the EntityDictionary class created with the Order type, another instance of the specialized EntityDictionary class (the one based on object references) is instantiated and the pointers contained therein are set to reference a Customer type specifically. This implementation of generics greatly reduces code bloat by ensuring that the compiler creates only one specialized class for generic classes of reference types.  Even though the runtime uses the same internal generic type definition when the type parameter on a generic reference type varies, this behavior is superseded if the type parameter is a value type. Dictionary<int, Customer>, Dictionary<Guid, Order>, and Dictionary<long, Order> will require new internal type definitions, for example.   |  | | --- | | Language Contrast: Java—Generics  The implementation of generics in Java occurs entirely within the compiler, not within the Java Virtual Machine. Sun Microsystems, which originally developed Java (long before Oracle took it over), adopted this approach to ensure that no updated Java Virtual Machine would need to be distributed because generics were used.  The Java implementation uses syntax like the templates in C++ and the generics in C#, including type parameters and constraints. Because it does not treat value types differently from reference types, however, the unmodified Java Virtual Machine cannot support generics for value types. As such, generics in Java do not offer the same gains in execution efficiency as they do in C#. Indeed, whenever the Java compiler needs to return data, it injects automatic downcasts from the specified constraint, if one is declared, or the base Object type, if a constraint is not declared. Further, the Java compiler generates a single specialized type at compile time, which it then uses to instantiate any constructed type. Finally, because the Java Virtual Machine does not support generics natively, there is no way to ascertain the type parameter for an instance of a generic type at execution time, and other uses of reflection are severely limited. | |
|  | |

\*\*\*COMP: End Advanced Topic after Language Contrast

Summary

The addition of generic types and methods to C# 2.0 fundamentally transformed the coding style of C# developers. In almost all cases in which programmers used object within C# 1.0 code, generics became a better choice in C# 2.0. In modern C# programs, using object (particularly in the context of any collection type) should make you consider whether the problem would be better solved with generics. The increased type safety enabled by elimination of casts, the elimination of the boxing performance penalty, and reduction of repeated code are all significant improvements.

Chapter 15 looks more at the most pervasive generic namespaces, System.Collections.Generic. As its name implies, this namespace is composed almost exclusively of generic types. It provides clear examples of how some types that originally used objects were then converted to use generics. However, before we tackle these topics, we will investigate expressions, which provide a significant C# 3.0 (and later) improvement for working with collections.

\*\*\*COMP: Insert “End 2.0” tab