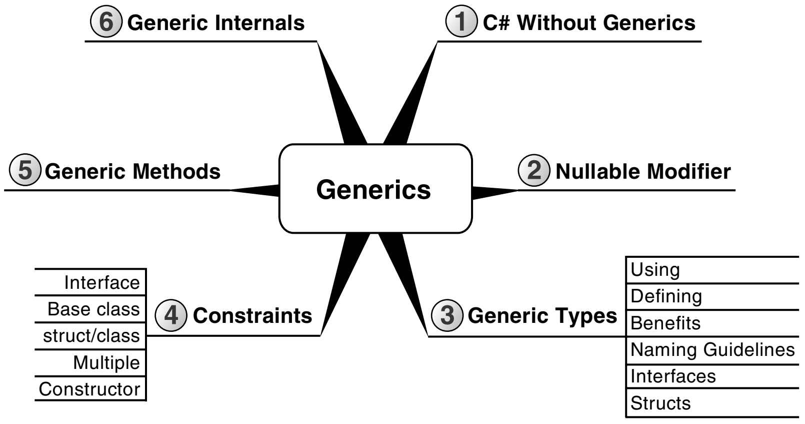
11 Generics

AS 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**. Just as methods are powerful because they can take parameters, classes that take type parameters have significantly more functionality as well, and this is what generics enable. Like their predecessor, templates, generics enable the definition of algorithms and pattern implementations once, rather than separately for each type. However, C# implements a type-safe version of templates that differs slightly in syntax and greatly in implementation from its predecessors in C++ and Java. Note that generics were added to the runtime and C# with version 2.0.



C# without Generics

I will begin the discussion of generics by examining a class that does not use generics. The class is `System.Collections.Stack`, and its purpose is to represent a collection of objects such that the last item to be added to the collection is the first item retrieved from the collection (called last in, first out, or 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 `Pop()` and `Push()` methods on the stack class appear in Listing 11.1.

LISTING 11.1: The `Stack` Definition Using a Data Type Object

```
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 11.2 uses the `stack` class for undo operations within a program which simulates the Etch A Sketch game.

LISTING 11.2: Supporting Undo in a Program Similar to the Etch A Sketch Game

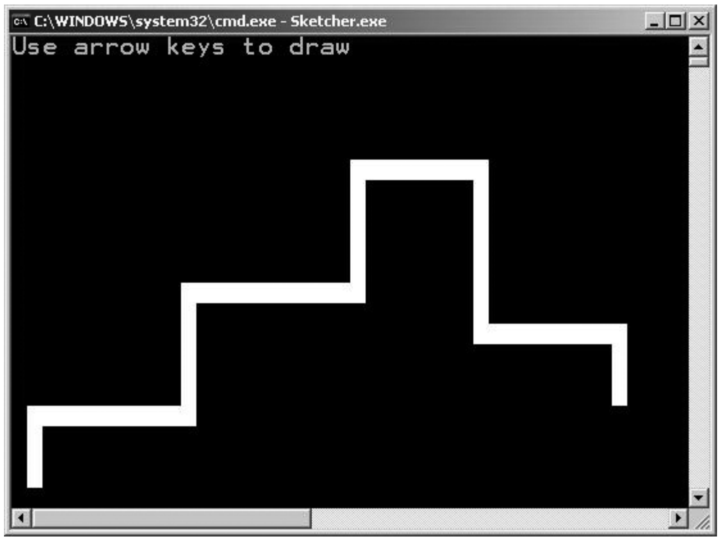
```
using System;
using System.Collections;

class Program
{
  // ...

  public void Sketch()
  {
      Stack path = new Stack();
      Cell currentPosition;
      ConsoleKeyInfo key;  // New with 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();  // New with C#2.0
                    break;
            }

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

    }
}

public struct Cell
{
    readonly public int X;
    readonly public int Y;
    public Cell(int x, int y)
    {
        X = x;
        Y = y;
    }
}
```

The results of Listing 11.2 appear in Output 11.1.

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 `Ctrl+Z`), then you undo the previous move

OUTPUT 11.1:



```
Use arrow keys to draw
```

by retrieving it from the stack using a `Pop()` method, setting the cursor position to be the previous position, and calling `Undo()`. (Note that this code uses some CLR 2.0-specific console functions as well.)

Although the code is functional, there is a fundamental drawback in the `System.Collections.Stack` class. As shown in Listing 11.1, the `Stack` class collects variables of type `object`. Because every object in the 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 in some scenarios, it is desirable.

Furthermore, when retrieving the data from the stack using the `Pop()` method, you must cast the return value to a `Cell`. But if the value returned from the `Pop()` method is not a `Cell` type object, an exception is thrown. You can test the data type, but splattering such checks builds complexity. The fundamental problem with creating classes that can work with multiple data types without generics is that they must use a common base type, generally `object` data.

Using value types, such as a struct or 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. While the widening operation (cast to a base class) for a reference type has a negligible performance impact, the box operation for a value type introduces nontrivial overhead.

To change the `Stack` class to enforce storage on a particular data type using the preceding C# programming constructs, you must create a specialized stack class, as in Listing 11.3.

LISTING 11.3: 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.

BEGINNER TOPIC

Another Example: Nullable Value Types

Chapter 2 introduced the capability of declaring variables that could contain `null` by using the nullable modifier, `?`, when declaring the value type variable. C# only began supporting this 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 11.4.

LISTING 11.4: Declaring Versions of Various Value Types That Store `null`

```
struct NullableInt
{
    public int Value
    {...}
    public bool HasValue
    {...}
    ...
}

struct NullableGuid
{
    public Guid Value
    {...}
    public bool HasValue
    {...}
    ...
}
...
```

Listing 11.4 shows implementations for only `NullableInt` and `NullableGuid`. If a program required additional nullable value types, you would have to create a copy with the additional value type. If the nullable implementation changed (if it supported a cast from a null to the nullable type, for example), you would have to add the modification to all of the nullable type declarations.

The second option was to declare a nullable type that contains a `Value` property of type `object`, as shown in Listing 11.5.

LISTING 11.5: Declaring a Nullable Type That Contains a `Value` Property of Type `object`

```
struct Nullable
{
    public object Value
    {...}
    public bool HasValue
    {...}
    ...
}
```

Although this option requires only one implementation of a nullable type, the runtime always boxes value types when setting the `Value` property. Furthermore, calls to retrieve data from `Nullable.Value` will not be strongly typed, so retrieving the value type will require a cast operation, which is potentially invalid at runtime.

Neither option is particularly attractive. To deal with dilemmas like this, C# 2.0 includes the concept of generics. In fact, the nullable modifier, `?`, uses generics internally.

Introducing Generic Types

Generics provide a facility for creating data structures that are specialized to handle specific types when declaring a variable. 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 based on programmer preference.

To minimize the learning curve for developers, C# designers chose syntax that matched the similar templates concept of C++. In C#, therefore, the syntax for generic classes and structures uses the same angle bracket notation to identify the data types on which the generic declaration specializes.

Using a Generic Class

Listing 11.6 shows how you can specify the actual type used by the generic class. You instruct the `path` variable to use a `Cell` type by specifying `Cell` within angle bracket notation in both the instantiation and the declaration expressions. 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. An example showing the new `Stack` class appears in Listing 11.6.

LISTING 11.6: Implementing Undo with a Generic `Stack` Class

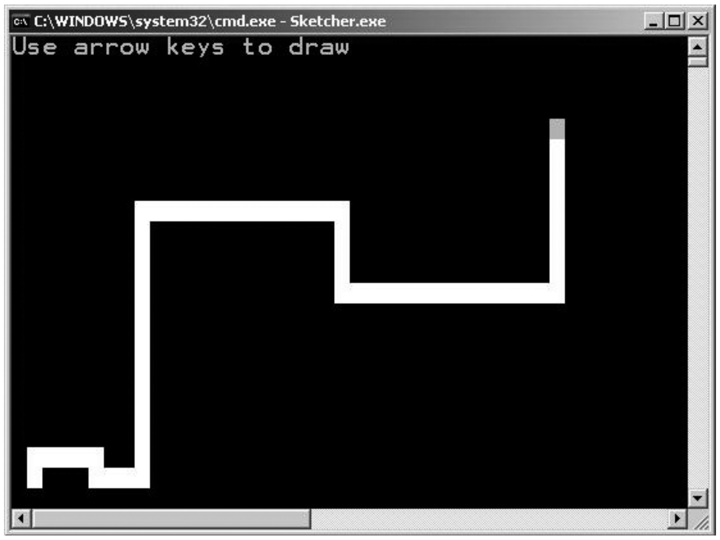
```
using System;
using System.Collections.Generic;

class Program
{
  // ...

  public void Sketch()
  {
      Stack<Cell> path;          // Generic variable declaration
      path = new Stack<Cell>();  // Generic object instantiation
      Cell currentPosition;
      ConsoleKeyInfo key;         // New with C# 2.0

      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();  // New with C#2.0
                    break;
            }

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

The results of Listing 11.6 appear in Output 11.2.

In the `path` declaration shown in Listing 11.6, you declare and create a new instance of a `System.Collections.Generic.Stack<T>` class and specify in angle brackets that the data type used for the `path` variable is `Cell`. As a result, every object added to and retrieved from `path` is of type `Cell`. In other words, 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. Before examining the generic advantages, the next section introduces the syntax for generic class definitions.

OUTPUT 11.2:



```
Use arrow keys to draw
```

Defining a Simple Generic Class

Generics allow you to author algorithms and patterns, and reuse the code for different data types. Listing 11.7 creates a generic `Stack<T>` class similar to the `System.Collections.Generic.Stack<T>` class used in the code in Listing 11.6. You specify a **type parameter identifier** or **type parameter** (in this case, `T`) within angle brackets after the class declaration. Instances of the generic `Stack<T>` then collect the type corresponding to the variable declaration without converting the collected item to type `object`. The type parameter `T` is a placeholder until variable declaration and instantiation, when the compiler requires the code to specify the type parameter. In Listing 11.7, 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 11.7: Declaring a Generic Class, **`Stack<T>`**

```
public class Stack<T>
{
    private T[] _Items;

    public void Push(T data)
    {
        ...
    }
```

```
        public T Pop()
        {
            ...
        }
    }
```

Benefits of Generics

There are several advantages to using a generic class (such as the `System.Collections.Generic.Stack<T>` class used earlier instead of the original `System.Collections.Stack` type).

1. Generics facilitate a strongly typed programming model, preventing data types other than those explicitly intended by the members within the parameterized class. In Listing 11.7, the parameterized stack class restricts you to the `Cell` data type for all instances of `Stack<Cell>`. (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 `garbage`, 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 cast to `object`; they no longer require a boxing operation. (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 increases because casting from an object is no longer required, thus eliminating a type check operation. Also, performance increases because boxing is no longer necessary for value types.
6. Generics reduce memory consumption because the avoidance of boxing no longer consumes 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 you name a method parameter, you should be as descriptive as possible when naming a type parameter. Furthermore, to distinguish it as being a type parameter, the name should include a "T" prefix. For example, in defining a class such as `EntityCollection<TEntity>` you use the type parameter name "TEntity."

The only time you would not use a descriptive type parameter name is when the 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 a type name of "TComponent."

Generic Interfaces and Structs

C# 2.0 supports the use of generics in all parts of the C# language, including interfaces and structs. The syntax is identical to that used by classes. To define an interface with a type parameter, place the type parameter in angle brackets, as shown in the example of `IPair<T>` in Listing 11.8.

Listing 11.8: 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. However, implementing a generic interface without identifying the type parameter forces the class to be a generic class, as shown in Listing 11.9. In addition, this example uses a struct rather than a class, indicating that C# supports custom generic value types.

LISTING 11.9: Implementing a Generic Interface

```
public struct Pair<T>: IPair<T>
{
    public T First
    {
        get
        {
            return _First;
        }
        set
        {
            _First = value;
        }
    }
    private T _First;

    public T Second
    {
        get
        {
            return _Second;
        }
        set
        {
            _Second = value;
        }
    }
    private T _Second;
}
```

Support for generic interfaces is especially important for collection classes, where generics are most prevalent. Without generics, 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 generic interfaces, you can avoid cast operations, because a stronger compile-time binding can be achieved with parameterized interfaces.

ADVANCED TOPIC

Implementing the Same Interface Multiple Times on a Single Class

One side effect of template interfaces is that you can implement the same interface many times using different type parameters. Consider the `IContainer<T>` example in Listing 11.10.

LISTING 11.10: 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 is not possible, and instead, the compiler would allow only one explicit `IContainer.Items` property.

One possible improvement on `Person` would be to also implement `IContainer<object>` and to have items return the combination of all three containers (`Address`, `Phone`, and `Email`).

Defining a Constructor and a Finalizer

Perhaps surprisingly, the constructor and destructor on a generic do not require type parameters in order to match the class declaration (i.e., not `Pair<T>(){...}`). In the pair example in Listing 11.11, the constructor is declared using `public Pair(T first, T second)`.

Listing 11.11: 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{ return _First; }
      set{ _First = value; }
  }
  private T _First;

  public T Second
  {
      get{ return _Second; }
      set{ _Second = value; }
  }
  private T _Second;
}
```

Specifying a Default Value

Listing 11.1 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. 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 11.12, causes a compile error because the field `_Second` goes uninitialized at the end of the constructor. Providing initialization for `_Second` presents a problem since you don't know the data type of `T`. If it is a reference type, then `null` would work, but this would not suffice if `T` were a value type (unless it was nullable).

Listing 11.12: Not Initializing All Fields, Causing a Compile 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;
  // }

  // ...
}
```

To deal with this scenario, C# 2.0 allows a dynamic way to code the default value of any data type using the `default` operator, first discussed in Chapter 8. In Chapter 8, I showed how the default value of `int` could be specified with `default(int)` while the default value of a string uses `default(string)` (which returns `null`, as it would for all reference types). In the case of `T`, which `_Second` requires, you use `default(T)` (see Listing 11.13).

Listing 11.13: Initializing a Field with the `default` Operator

```
public struct Pair<T>: IPair<T>
{
  public Pair(T first)
  {
      _First = first;
      _Second = default(T);
  }
```

```
    // ...
  }
```

The `default` operator is allowable outside of the context of generics; any statement can use it.

Multiple Type Parameters

Generic types may employ any number of type parameters. The initial `Pair<T>` example contains only one type parameter. To enable support for storing a dichotomous pair of objects, such as a name/value pair, you need to extend `Pair<T>` to support two type parameters, as shown in Listing 11.14.

LISTING 11.14: 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{ return _First;    }
        set{ _First = value; }
    }
    private TFirst _First;

    public TSecond Second
    {
        get{ return _Second; }
        set{ _Second = value; }
    }
    private TSecond _Second;
}
```

When you use the `Pair<TFirst, TSecond>` class, you supply multiple type parameters within the angle brackets of the declaration and instantiation statements, and then you supply matching types to the parameters of the methods when you call them, as shown in Listing 11.15.

LISTING 11.15: 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, the **arity**, uniquely distinguishes the class. Therefore, it is possible to define both `Pair<T>` and `Pair<TFirst, TSecond>` within the same namespace because of the arity variation.

Nested Generic Types

Nested types will automatically inherit the type parameters of the containing type. If the containing type includes a type parameter `T`, for example, then the type `T` will be available on the nested type as well. If the nested type includes its own type parameter named `T`, then 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 11.16).

LISTING 11.16: Nested Generic Types

```
class Container<T, U>
{
  // Nested classes inherit type parameters.
  // Reusing a type parameter name will cause
  // a warning.
  class Nested<U>
  {
      void Method(T param0, U param1)
      {
      }
  }
}
```

The behavior of making the container's type parameter available in the nested type is consistent with nested type behavior in the sense that private members of the containing type are also accessible from the nested type. The rule is simply that a type is available anywhere within the curly braces within which it appears.

Type Compatibility between Generic Classes with Type-Compatible Type Parameters

If you declare two variables with different type parameters using the same generic class, the variables are not type compatible; they are not **covariant**. The type parameter differentiates two variables of the same generic class but with different type parameters. For example, instances of a generic class, `Stack<Contact>` and `Stack<PdaItem>`, are not type compatible even when the type parameters are compatible. In other words, there is no built-in support for casting `Stack<Contact>` to `Stack<PdaItem>`, even though `Contact` derives from `PdaItem` (see Listing 11.17).

LISTING 11.17: Conversion between Generics with Different Type Parameters

```
using System.Collections.Generic;
...
// Error: Cannot convert type ...
Stack<PdaItem> exceptions = new Stack<Contact>();
```

To allow this you would have to subtly cast each instance of the type parameter, possibly an entire array or collection, which would hide a potentially significant performance cost.

Constraints

Generics support the ability to define constraints on type parameters. These constraints enforce the types to conform to various rules. Take, for example, the `BinaryTree<T>` class shown in Listing 11.18.

LISTING 11.18: Declaring a **`BinaryTree<T>`** Class with No Constraints

```
public class BinaryTree<T>
{
    public BinaryTree ( T item)
    {
        Item = item;
        }

        public T Item
        {
            get{ return _Item;    }
            set{ _Item = value; }
        }
        private T _Item;

        public Pair<BinaryTree<T>> SubItems
        {
            get{ return _SubItems; }
            set{ _SubItems = value; }
        }
        private Pair<BinaryTree<T>> _SubItems;
    }
```

(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. In order to achieve the sorting, the `SubItems` get accessor uses the `CompareTo()` method of the supplied key, as shown in Listing 11.19.

LISTING 11.19: Needing the Type Parameter to Support an Interface

```
    public class BinaryTree<T>
    {
        ...
        public Pair<BinaryTree<T>> SubItems
        {
            get{ return _SubItems; }
            set
            {
                IComparable first;
                // ERROR: Cannot implicitly convert type...
                first = value.First.Item;  // Explicit cast required

                if (first.CompareTo(value.Second.Item) < 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 generic. Written as is, the compiler assumes that the only members available on `T` are those inherited from the base type `object`, since every type has `object` as an ancestor. (Only methods such as `ToString()`, therefore, are available to the key 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` interface in order to access the `CompareTo()` method, as shown in Listing 11.20.

LISTING 11.20: Needing the Type Parameter to Support an Interface or Exception Thrown

```
    public class BinaryTree<T>
    {
        ...
        public Pair<BinaryTree<T>> SubItems
        {
            get{ return _SubItems; }
            set
            {
                IComparable first;
                first = (IComparable)value.First.Item;

                if (first.CompareTo(value.Second.Item) < 0)
                {
                    // first is less than second.
                    ...
                }
                else
                {
                    // second is less than or equal to first.
                    ...
                }
                _SubItems = value;
            }
        }
        private Pair<BinaryTree<T>> _SubItems;
    }
```

Unfortunately, however, if you now declare a `BinaryTree` class variable and supply a type parameter that does not implement the `IComparable` michaelis.book Page 399 Thursday, June 15, 2006 3:09 PM

interface, you encounter an execution-time error—specifically, an `InvalidCastException`. This defeats an advantage of generics.

Language Contrast: C++—Templates

Generics in C# and the CLR differ from similar constructs in other languages. While other languages provide similar functionality, C# is significantly more type safe. Generics in C# is a language feature and a platform feature, the underlying 2.0 runtime contains deep support for generics in its engine.

C++ templates differ significantly from C# generics, because C# takes advantage of the CIL. C# generics are compiled into CIL, causing specialization to occur at execution time for each value type only when it is used, and only once for reference types.

A distinct feature not supported by C++ templates is explicit constraints. C++ templates allow you to compile a method call that may or may not belong to the type parameter. As a result, if the member does not exist in the type parameter, an error occurs, likely with a cryptic error message and referring to an unexpected location in the source code. However, the advantage of the C++ implementation is that operators (+, -, and so on) may be called on the type. C# does not support the calling of operators on the type parameter because operators are static—so they can't be identified by interfaces or base class constraints.

The problem with the error is that it occurs only when *using* the template, not when defining it. Because C# generics can declare constraints, the compiler can prevent such errors when defining the generic, thereby identifying invalid assumptions sooner. Furthermore, when declaring a variable of a generic type, the error will point to the declaration of the variable, not to the location in the generic implementation where the member is used.

It is interesting to note that Microsoft's CLI support in C++ includes both generics and C++ templates because of the distinct characteristics of each.

To avoid this exception and instead provide a compile-time error, C# enables you to supply an optional list of **constraints** for each type parameter declared in the generic class. A constraint declares the type parameter characteristics that the generic requires. You declare a constraint using the `where` keyword, followed by a "parameter-requirements" pair, where the parameter must be one of those defined in the generic type and the requirements are to restrict the class or interface from which the type "derives," the presence of a default constructor, or a reference/value type restriction.

Interface Constraints

In order to satisfy the sort requirement, you need to use the `CompareTo()` method in the `BinaryTree` class. To do this most effectively, you impose a constraint on the `T` type parameter. You need the `T` type parameter to implement the `IComparable` interface. The syntax for this appears in Listing 11.21.

LISTING 11.21: Declaring an Interface Constraint

```
public class BinaryTree<T>
    where T: System.IComparable
{
    ...
    public Pair<BinaryTree<T>> SubItems
    {
        get{ return _SubItems; }
        set
        {
            IComparable first;
            // Notice that the cast can now be eliminated.
            first = value.First.Item;

            if (first.CompareTo(value.Second.Item) < 0)
            {
                // first is less than second
                ...
            }
            else
            {
                // second is less than or equal to first.
                ...
            }
            _SubItems = value;
        }
    }
    private Pair<BinaryTree<T>> _SubItems;
}
```

Given the interface constraint addition in Listing 11.21, the compiler ensures that each time you use the `BinaryTree` class you specify a type parameter that implements the `IComparable` interface. Furthermore, you no longer need to explicitly cast the variable to an `IComparable` 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. To resolve what member to call, the compiler first checks class members directly, and then looks at the explicit interface members. If no constraint resolves the argument, only members of `object` are allowable.

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`. The error is similar to the one shown in Output 11.3.

OUTPUT 11.3:

```
error CS0309: The type 'System.Text.StringBuilder>' must be convertible
to 'System.IComparable' in order to use it
as parameter 'T' in the generic type or method 'BinaryTree<T>'
```

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

Base Class Constraints

Sometimes you might want to limit the constructed type to a particular class derivation. You do this using a **base class constraint**, as shown in Listing 11.22.

LISTING 11.22: Declaring a Base Class Constraint

```
public class EntityDictionary<TKey, TValue>
    : System.Collections.Generic.Dictionary<TKey, TValue>
    where TValue : EntityBase
{
    ...
}
```

In contrast to `System.Collections.Generic.Dictionary<TKey, TValue>` on its own, `EntityDictionary<TKey, TValue>` requires that all `TValue` types derive from the `EntityBase` class. By requiring the derivation, it is possible to always perform a cast operation within the generic implementation, because the constraint will ensure that all type parameters derive from the base and, therefore, that all `TValue` type parameters used with `EntityDictionary` can be implicitly converted to the base.

The syntax for the base class constraint is the same as that for the interface constraint, except that base class constraints must appear first when multiple constraints are specified. However, unlike interface constraints, multiple base class constraints are not allowed since it is not possible to derive from multiple classes. Similarly, base class constraints cannot be specified for sealed classes or specific structs. For example, C# does not allow a constraint for a type parameter to be derived from `string` or `System.Nullable<T>`.

`struct/class` Constraints

Another valuable generic constraint is the ability to restrict type parameters to a value type or a reference type. The compiler does not allow specifying `System.ValueType` as the base class in a constraint. Instead, C# provides special syntax that works for reference types as well. Instead of specifying a class from which `T` must derive, you simply use the keyword `struct` or `class`, as shown in Listing 11.23.

LISTING 11.23: Specifying the Type Parameter as a Value Type

```
public struct Nullable<T> :
     IFormattable, IComparable,
     IComparable<Nullable<T>>, INullable
     where T : struct
{
    // ...
}
```

Because a base class constraint requires a particular base class, using `struct` or `class` with a base class constraint would be pointless, and in fact could allow for conflicting constraints. Therefore, you cannot use `struct` and `class` constraints with a base class constraint.

There is one special characteristic for the `struct` constraint. It limits possible type parameters as being only value types while at the same time preventing type parameters that are `System.Nullable<T>` type parameters. Why? Without this last restriction, it would be possible to define the nonsense type `Nullable<Nullable<T>>`, which is nonsense because `Nullable<T>` on its own allows a value type variable that supports nulls, so a nullable-nullable type becomes meaningless. Since the nullable operator (`?`) is a C# shortcut for declaring a nullable value type, the `Nullable<T>` restriction provided by the `struct` constraint also prevents code such as the following:

```
int?? number   // Equivalent to Nullable<Nullable<int> if allowed
```

Multiple Constraints

For any given type parameter, you may specify any number of interfaces as constraints, but no more than one class, 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 and a colon. If there is more than one type parameter, each must be preceded by the `where` keyword. In Listing 11.24, the `EntityDictionary` class contains two type parameters: `TKey` and `TValue`. The `TKey` type parameter has two interface constraints, and the `TValue` type parameter has one base class constraint.

LISTING 11.24: Specifying Multiple Constraints

```
public class EntityDictionary<TKey, TValue>
    : Dictionary<TKey, TValue>
    where TKey : IComparable, 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. `TKey` must implement `IComparable` 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 a type parameter inside the generic class. In Listing 11.25, the `New()` method for the `EntityDictionary<TKey, TValue>` class must create an instance of the type parameter `TValue`.

LISTING 11.25: Requiring a Default Constructor Constraint

```
public class EntityBase<TKey>
{
    public TKey Key
    {
        get{ return _Key;  }
        set{ _Key = value; }
    }
    private TKey _Key;
}

public class EntityDictionary<TKey, TValue> :
    Dictionary<TKey, TValue>
    where TKey: IComparable, IFormattable
    where TValue : EntityBase<TKey>, new()
{
    // ...

    public TValue New(TKey key)
    {
        TValue newEntity = new TValue();
        newEntity.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 the type parameter. To override this compiler restriction, you add the text `new()` after all other constraints are specified. This text is a **constructor constraint**, and it forces the type parameter decorated with the constructor constraint to have a default constructor. Only the default constructor constraint is available. You cannot specify a constraint for a constructor with parameters.

Constraint Inheritance

Constraints are inherited by a derived class, but they must be specified explicitly on the derived class. Consider Listing 11.26.

Listing 11.26: Inherited Constraints Specified Explicitly

```
class EntityBase<T> where T : IComparable
{
}
```

```
// ERROR:
// The type 'T' must be convertible to 'System.IComparable'
// in order to use it as parameter 'T' in the generic type or
// method.
// class Entity<T> : EntityBase<T>
// {
// }
```

Because `EntityBase` requires that `T` implement `IComparable`, the `Entity` class needs to explicitly include the same constraint. Failure to do so will result in a compile error. This increases a programmer's awareness of the constraint in the derived class, avoiding confusion when using the derived class and discovering the constraint, but not understanding where it comes from.

ADVANCED TOPIC

Constraint Limitations

Constraints are appropriately limited to avoid nonsense code. For example, you cannot combine a base class constraint with a `struct` or `class` constraint, nor can you use `Nullable<T>` on struct constraint type parameters. Also, you cannot specify constraints to restrict inheritance to special types such as `object`, arrays, `System.ValueType`, `System.Enum` (enum), `System.Delegate`, and `System.MulticastDelegate`.

In some cases, constraint limitations are perhaps more desirable, but they still are not supported. The following subsections provide some additional examples of constraints that are not allowed.

Operator Constraints Are Not Allowed

Another restriction on constraints is that you cannot specify a constraint that a class supports on a particular method or operator, unless that method or operator is on an interface. Because of this, the generic `Add()` in Listing 11.27 does not work.

LISTING 11.27: 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. However, because all types support only the methods of `object` (which does not include the + operator), an error occurs. Unfortunately, there is no way to specify the + operator within a constraint; therefore, creating an add method like this is a lot more cumbersome. One reason for this limitation is that there is no way to constrain a type to have a static method. You cannot, for example, specify static methods on an interface.

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 TKey : IComparable, IFormattable` requires that both `IComparable` and `IFormattable` are supported. There is no way to specify an OR relationship between constraints. Hence, an equivalent of Listing 11.28 is not supported.

LISTING 11.28: Combining Constraints Using an OR Relationship Is Not Allowed

```
public class BinaryTree<T>
    // Error:  OR is not supported.
    where T: System.IComparable || System.IFormattable<T>
{
    ...
}
```

Supporting this would prevent the compiler from resolving which method to call at compile time.

Constraints of Type Delegate and Enum Are Not Valid

Readers who are already familiar with C# 1.0 and are reading this chapter to learn 2.0 features will be familiar with the concept of delegates, which are covered in Chapter 13. One additional constraint that is not allowed is the use of any delegate type as a class constraint. For example, the compiler will output an error for the class declaration in Listing 11.29.

LISTING 11.29: Inheritance Constraints Cannot Be of Type `System.Delegate`

```
// Error:  Constraint cannot be special class 'System.Delegate'
public class Publisher<T>
    where T : System.Delegate
{
    public event T Event;
    public void Publish()
    {
        if (Event != null)
        {
            Event(this, new EventArgs());
        }
    }
}
```

All delegate types are considered special classes that cannot be specified as type parameters. Doing so would prevent compile-time validation on the call to `Event()` because the signature of the event firing is unknown with the data types `System.Delegate` and `System.MulticastDelegate`. The same restriction occurs for any enum type.

Constructor Constraints Are Allowed Only for Default Constructors

Listing 11.25 includes a constructor constraint that forces `TValue` to support a default constructor. There is no constraint to force `TValue` to support a constructor other than the default. For example, it is not possible to make `EntityBase.Key` protected and only set it in a `TValue` constructor that takes a `TKey` parameter using constraints alone. Listing 11.30 demonstrates the invalid code.

LISTING 11.30: 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 11.31).

LISTING 11.31: Using a Factory Interface in Place of a Constructor Constraint

```
public class EntityBase<TKey>
{
  public EntityBase(TKey key)
  {
      Key = key;
  }
  public TKey Key
  {
      get { return _key; }
      set { _key = value; }
  }
  private TKey _key;
}

public class EntityDictionary<TKey, TValue, TFactory> :
      Dictionary<TKey, TValue>
  where TKey : IComparable, IFormattable
  where TValue : EntityBase<TKey>
  where TFactory : IEntityFactory<TKey, TValue>, new()
{
  ...
  public TValue New(TKey key)
  {
      TValue newEntity = new TFactory().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` constructor that takes parameters rather than the default constructor. It no longer uses the constructor constraint on `TValue` because `TFactory` is responsible for instantiating the order instead of `EntityDictionary<...>`. (One modification to the code in Listing 11.31 would be to save a copy of the factory. This would enable you to reuse the factory 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 11.32.

LISTING 11.32: 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);
  }
}
```

Generic Methods

You already learned that it is relatively simple to add a generic method to a class when the class is a generic. You did this in the generic class examples so far, and it also works for static methods. Furthermore, you can use generic classes within a generic class, as you did in earlier `BinaryTree` listings using the following line of code:

```
public Pair< BinaryTree<T> > SubItems;
```

Generic methods are methods that use generics even when the containing class is not a generic class or the method contains type parameters not included in the generic class type parameter list. To define generic methods, you add the type parameter syntax immediately following the method name, as shown in the `MathEx.Max<T>` and `MathEx.Min<T>` examples in Listing 11.33.

LISTING 11.33: Defining Generic Methods

```
public static class MathEx
{
  public static T Max<T>(T first, params T[] values)
      where T : IComparable
  {
      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 minimum = first;

        foreach (T item in values)
        {
            if (item.CompareTo(minimum) < 0)
            {
                minimum = item;
            }
        }
      return minimum;
  }
}
```

You use the same syntax on a generic class when the method requires an additional type parameter not included in the class type parameter list. In this example, the method is static but C# does not require this.

Note that generic methods, like classes, can include more than one type parameter. The arity (the number of type parameters) is an additional distinguishing characteristic of a method signature.

Type Inferencing

The code used to call the `Min<T>` and `Max<T>` methods looks like that shown in Listing 11.34.

Listing 11.34: Specifying the Type Parameter Explicitly

```
Console.WriteLine(
    MathEx.Max<int>(7, 490));
Console.WriteLine(
    MathEx.Min<string>("R.O.U.S.", "Fireswamp"));
```

The output to Listing 11.34 appears in Output 11.4.

Output 11.4:

```
490
Fireswamp
```

Not surprisingly, the type parameters, `int` and `string`, correspond to the actual types used in the generic method calls. However, specifying the type is redundant because the compiler can infer the type from the parameters passed to the method. To avoid redundancy, you can exclude the type parameters from the call. This is known as **type inferencing**, and an example appears in Listing 11.35. The output of this listing appears in Output 11.5.

Listing 11.35: Inferring the Type Parameter

```
Console.WriteLine(
    MathEx.Max(7, 490));
Console.WriteLine(
    MathEx.Min("R.O.U.S'", "Fireswamp"));
```

Output 11.5:

```
490
Fireswamp
```

For type inferencing to be successful, the types must match the method signature. Calling the `Max<T>` method using `MathEx.Max(7.0, 490)`, for example, causes a compile error. You can resolve the error by either casting explicitly or including the type argument. Also note that you cannot perform type inferencing purely on the return type. Parameters are required for type inferencing to be allowed.

Specifying Constraints

The generic method also allows constraints to be specified. For example, you can restrict a type parameter to implement `IComparable`. The constraint is specified immediately following the method header, prior to the curly braces of the method block, as shown in Listing 11.36.

Listing 11.36: 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
    {
        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);
    }
}
```

Notice that the `Show<T>` implementation itself does not use the `IComparable` interface. Recall, however, that the `BinaryTree<T>` class did require this (see Listing 11.37).

Listing 11.37: `BinaryTree<T>` Requiring `IComparable` Type Parameters

```
public class BinaryTree<T>
    where T: System.IComparable
{
    ...
}
```

Because the `BinaryTree<T>` class requires this constraint on `T`, and because `Show<T>` uses `BinaryTree<T>`, `Show<T>` also needs to supply the constraint.

ADVANCED TOPIC

Casting inside a Generic Method

Sometimes you should be wary of using generics—for instance, when using it specifically to bury a cast operation. Consider the following method, which converts a stream into an object:

```
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 user of the method, `Deserialize<T>()` appears to be strongly typed. However, a cast operation is still performed implicitly rather than explicitly, as in the case of the nongeneric equivalent shown here:

```
string greeting =
    (string)Deserialization.Deserialize(stream, formatter);
```

A method using an explicit cast is more explicit about what is taking place than is a generic version with a hidden cast. Developers should use care when casting in generic methods if there are no constraints to verify cast validity.

Generic Internals

Given the discussions in earlier chapters about the prevalence of objects within the CLI type system, it is no surprise 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. So, the addition of generics to the C# language is a feature of both the compiler and the platform. 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 no different from a regular class. The result of the compilation is nothing but metadata and CIL. The CIL is parameterized to accept a user-supplied type somewhere in code. Suppose you had a simple `Stack` class declared as shown in Listing 11.38.

Listing 11.38: `Stack<T>` Declaration

```
public class Stack<T> where T : IComparable
{
    T[] items;
    // rest of the class here

}
```

When you compile the class, the generated CIL is parameterized and looks something like Listing 11.39.

Listing 11.39: 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. It declares the number of parameter types that the generic class will include. A declaration such as `EntityDictionary<TKey, TValue>` would have an arity of 2.

In addition, 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," new to 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 11.40.

LISTING 11.40: 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.

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 11.41.

LISTING 11.41: `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 `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 11.42, you declare two instances of a `Stack<int>`, both using the code already generated by the runtime for a `Stack<int>`.

LISTING 11.42: 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 as its type parameter (such as a `long` or a user-defined `struct`), the runtime generates another version of the generic type. The benefit of specialized value type classes is better performance. Furthermore, the code is able to avoid conversions and boxing because each specialized generic class "natively" contains the value type.

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, not a specialized generic type based on the type parameter. 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, and you create an `EntityDictionary` of `Customer` types, like so:

```
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 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.

Language Contrast: Java—Generics

Sun's implementation of generics for Java occurs within the compiler entirely, not within the Java Virtual Machine. Sun did this to ensure that no updated Java Virtual Machine would need to be distributed because generics were used.

The Java implementation uses syntax similar to the templates in C++ and generics in C#, including type parameters and constraints. But because it does not treat value types differently from reference types, the unmodified Java Virtual Machine cannot support generics for value types. As such, generics in Java do not gain the execution efficiency of 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 it 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.

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 reducing to one the number of specialized classes created by the compiler 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.

SUMMARY

Generics will significantly transform C# 1.0 coding style. In virtually all cases in which programmers used `object` within C# 1.0 code, generics would be a better choice in C# 2.0 to the extent that `object` should act as a flag for a possible generics implementation. The increased type safety, cast avoidance, and reduction of code bloat offer significant improvements. Similarly, where code traditionally used the `System.Collections` namespace, `System.Collections.Generics` should be selected instead.

The next chapter looks at one of the most pervasive generic namespaces, `System.Collections.Generic`. 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.