Generics

C# has two separate mechanisms for writing code that is reusable across different types: *inheritance* and *generics*. Whereas inheritance expresses reusability with a base type, generics express reusability with a "template" that contains "placeholder" types. Generics, when compared to inheritance, can *increase type safety* and *reduce casting and boxing*.

NOTE
C# generics and C++ templates are similar concepts, but they work differently. We explain this difference in C# Generics Versus C++ Templates.

Generic Types

A generic type declares *type parameters*—placeholder types to be filled in by the consumer of the generic type, which supplies the *type arguments*. Here is a generic type Stack<T>, designed to stack instances of type T. Stack<T> declares a single type parameter T:

```
public class Stack<T>
{
int position;
T[] data = new T[100];
public void Push (T obj) {

data[position++] = obj; }
public T Pop() { return
data[--position]; }
}
```

We can use Stack<T> as follows:

```
Stack<int> stack = new Stack<int>();
stack.Push(5);
stack.Push(10);
int x = stack.Pop(); // x is 10
int y = stack.Pop(); // y is 5
```

Stack<int> fills in the type parameter T with the type argument int, implicitly creating a type on the fly (the synthesis occurs at runtime). Stack<int> effectively has the following definition (substitutions appear in bold, with the class name hashed out to avoid confusion):

```
public class ###
{
int position;
int[] data;
public void Push (int obj) {
data[position++] = obj; }

public int Pop() { return
data[--position]; }
}
```

Technically, we say that Stack<T> is an *open type*, whereas Stack<int> is a *closed type*. At runtime, all generic type instances are closed—with the placeholder types filled in. This means that the following statement is illegal:

```
var stack = new Stack<T>(); // Illegal:
What is T?
unless inside a class or method which itself defines T as a type parameter:
public class Stack<T>
{
...
public Stack<T> Clone()
{
Stack<T> clone = new Stack<T>(); //
Legal
...
}
}
```

Why Generics Exist

Generics exist to write code that is reusable across different types. Suppose we needed a stack of integers, but we didn't have generic types. One solution would be to hardcode a separate version of the class for every required element type (e.g., IntStack, StringStack, etc.). Clearly, this would cause considerable code duplication. Another solution would be to write a stack that is generalized by using object as the element type:

```
public class ObjectStack
{
int position;
```

```
object[] data = new object[10];
public void Push (object obj) {
data[position++] = obj; }
public object Pop() { return
data[--position]; }
}
```

An ObjectStack, however, wouldn't work as well as a hardcoded IntStack for specifically stacking integers. Specifically, an ObjectStack would require boxing and downcasting that could not be checked at compile time:

```
// Suppose we just want to store integers
here:
ObjectStack stack = new ObjectStack();
stack.Push ("s"); // Wrong type,but no error!
int i = (int)stack.Pop(); // Downcast -runtime error
```

What we need is both a general implementation of a stack that works for all element types, and a way to easily specialize that stack to aspecific element type for increased type safety and reduced casting and boxing. Generics give us precisely this, by allowing us to parameterize the element type. Stack<T> has the benefits of both ObjectStack and IntStack. Like ObjectStack, Stack<T> is written once to work *generally* across all types. Like IntStack, Stack<T> is *specialized* for a particular type—the beauty is that this type is T, which we substitute on the fly.

NOTE

ObjectStack is functionally equivalent to Stack<object>.

Generic Methods

A generic method declares type parameters within the signature of a method. With generic methods, many fundamental algorithms can be implemented in a generalpurpose way only. Here is a generic method that swaps the contents of two variables of any type T:

```
static void Swap<T> (ref T a, ref T b)
{
T temp = a;
a = b;
b = temp;
```

}
Swap<T> can be used as follows:
int x = 5;
int y = 10;
Swap (ref x, ref y);

Generally, there is no need to supply type arguments to a generic method, because the compiler can implicitly infer the type. If there is ambiguity, generic methods can be called with the type arguments as follows:
Swap**<int>** (ref x, ref y);

Within a generic *type*, a method is not classed as generic unless it *introduces* type parameters (with the angle bracket syntax). The Pop method in our generic stack merely uses the type's existing type parameter, T, and is not classed as a generic method. Methods and types are the only constructs that can introduce type parameters. Properties, indexers, events, fields, constructors, operators, and so on cannot declare type parameters, although they can partake in any type parameters already declared by their enclosing type. In our generic stack example, for instance, we could write an indexer that returns a generic item:

public T this [int index] { get { return data [index]; } }

Similarly, constructors can partake in existing type parameters, but not *introduce* them:
public Stack<T>() { } // Illegal

Declaring Type Parameters

Type parameters can be introduced in the declaration of classes, structs, interfaces, delegates (covered in Chapter 4), and methods. Other constructs, such as properties, cannot *introduce* a type parameter, but can *use* one.
For example, the property Value uses T:
public struct Nullable**<T>**
{
public **T** Value { get; }
}

A generic type or method can have multiple parameters. For example:

class Dictionary<TKey, TValue> {...}
To instantiate:
Dictionary<int,string> myDic = new Dictionary<int,string>();
Or:
var myDic = new Dictionary<int,string>();

Generic type names and method names can be overloaded as long as the number of type parameters is different. For example, the following two type names do not conflict:
class A<T> {}
class A<T1,T2> {}

NOTE
By convention, generic types and methods with a *single* type parameter typically name their parameter T, as long as the intent of the parameter is clear. When using *multiple* type parameters, each parameter is prefixed with T, but has a more descriptive name.

typeof and Unbound Generic Types

Open generic types do not exist at runtime: open generic types are closed as part of compilation. However, it is possible for an *unbound* generic type to exist at runtime—purely as a Type object. The only way to specify an unbound generic type in C# is with the typeof operator:
class A<T> {}
class A<T1,T2> {}
...

Type a1 = **typeof (A<>)**;// *Unbound* type (notice no type arguments).
Type a2 = **typeof (A<,>)**;// Use commas to indicate multiple type args.
Open generic types are used in conjunction with the Reflection API (Chapter 19).
You can also use the typeof operator to specify a closed type:
Type a3 = typeof (A<int,int>); or an open type (which is closed at runtime):

```
class B<T> {
void X() {
Type t = typeof (T);
} }
```

The default Generic Value

The default keyword can be used to get the default value given a generic type parameter. The default value for a reference type is null, and the default value for a value type is the result of bitwise-zeroing the value type's fields:

```
static void Zap<T> (T[] array)
{
for (int i = 0; i < array.Length; i++)
array[i] = default(T);
}
```

Generic Constraints

By default, a type parameter can be substituted with any type whatsoever. *Constraints* can be applied to a type parameter to require more specific type arguments.
These are the possible constraints:

```
where T : base-class // Base-class constraint
where T : interface // Interface constraint
where T : class // Reference-type constraint
where T : struct // Value-type constraint (excludes Nullable types)
where T : new() // Parameterless constructor constraint
where U : T // Naked type constraint
```

In the following example, GenericClass<T,U> requires T to derive from (or be identical to) SomeClass and implement Interface1, and requires U to provide a parameterless constructor:

```
class SomeClass {}
interface Interface1 {}
class GenericClass<T,U> where T : SomeClass, Interface1 where U :
new()
{...}
```

Constraints can be applied wherever type parameters are defined, in both methods and type definitions. A *base-class constraint* specifies that the type parameter must subclass (or match) a particular class; an *interface constraint* specifies that the type parameter must implement that interface.

These constraints allow instances of the type parameter to be implicitly converted to that class or interface. For example, suppose we want to write a generic Max method, which returns the maximum of two values. We can take advantage of the generic interface defined in the framework called Icomparable<T>: public interface IComparable<T> // Simplified version of interface
{
int CompareTo (T other);
}
CompareTo returns a positive number if this is greater than other. Using this interface as a constraint, we can write a Max method as follows (to avoid distraction, null checking is omitted):

static T Max <T> (T a, T b) where T : IComparable<T>
{
return a.CompareTo (b) > 0 ? a : b;
}
The Max method can accept arguments of any type implementing IComparable<T> (which includes most built-in types such as int and string):
int z = Max (5, 10); // 10
string last = Max ("ant", "zoo"); // zoo
The *class constraint* and *struct constraint* specify that T must be a reference type or (non-nullable) value type. A great example of
the struct constraint is the System.Nullable<T> struct (we will discuss this class in depth in Nullable Types in Chapter 4): struct Nullable<T> where T : struct {...}

The *parameterless constructor constraint* requires T to have a public parameterless constructor. If this constraint is defined, you can call new() on T:
static void Initialize<T> (T[] array) where T : new()
{
for (int i = 0; i < array.Length; i++)
array[i] = new T();
}

The *naked type constraint* requires one type parameter to derive from (or match) another type parameter. In this example, the method FilteredStack

returns another Stack, containing only the subset of elements where the type parameter U is of the type parameter T:

```
class Stack<T>
{
Stack<U> FilteredStack<U>() where U : T
{...}
}
```

Subclassing Generic Types

A generic class can be subclassed just like a nongeneric class. The subclass can leave the base class's type parameters open, as in the following example:

```
class Stack<T> {...}
class SpecialStack<T> : Stack<T> {...}
```

Or the subclass can close the generic type parameters with a concrete type:

```
class IntStack : Stack<int> {...}
```

A subtype can also introduce fresh type arguments:

```
class List<T> {...}
class KeyedList<T,TKey> : List<T> {...}
```

NOTE

Technically, *all* type arguments on a subtype are fresh: you could say that a subtype closes and then reopens the base type arguments. This means that a subclass can give new (and potentially more meaningful) names to the type arguments it reopens:

```
class List<T> {...}
class KeyedList<TElement,TKey> :
List<TElement> {...}
```

Self-Referencing Generic Declarations

A type can name *itself* as the concrete type

when closing a type argument:

```
public interface IEquatable<T> { bool Equals
(T obj); }
public class Balloon : IEquatable<Balloon>
{
public string Color { get; set; }
public int CC { get; set; }
public bool Equals (Balloon b)
{
if (b == null) return false;
return b.Color == Color && b.CC == CC;
}
}
```

The following are also legal:

```
class Foo<T> where T : IComparable<T> { ... }
class Bar<T> where T : Bar<T> { ... }
```

Static Data

Static data is unique for each closed type:

```
class Bob<T> { public static int Count; }
class Test
{
static void Main()
{
Console.WriteLine(++Bob<int>.Count); // 1
Console.WriteLine(++Bob<int>.Count); // 2
Console.WriteLine(++Bob<string>.Count); // 1
Console.WriteLine(++Bob<object>.Count); // 1
}
}
```

Type Parameters and Conversions

C#’s cast operator can perform several kinds of conversion, including:

▪ Numeric conversion
▪ Reference conversion
▪ Boxing/unboxing conversion
▪ Custom conversion (via operator overloading; see Chapter 4)

The decision as to which kind of conversion will take place happens at *compile time*, based on the known types of the operands. This createsan interesting scenario with generic type parameters, because the precise operand types are unknown at compile time. If this leads to ambiguity, the compiler generates an error. The most common scenario is when you want to perform a reference conversion:

```
StringBuilder Foo<T> (T arg)
{
if (arg is StringBuilder)
return (StringBuilder) arg; // Will not compile
...
}
```

Without knowledge of T's actual type, the compiler is concerned that you might have intended this to be a *custom conversion*. The simplest solution is to instead use the as operator, which is unambiguous because it cannot perform custom conversions:

```
StringBuilder Foo<T> (T arg)
{
StringBuilder sb = arg as StringBuilder;
if (sb != null) return sb;
...
}
```

A more general solution is to first cast to object. This works because conversions to/from object are assumed not to be custom conversions, but reference or boxing/unboxing conversions. In this case, StringBuilder is a reference type, so it has to be a reference conversion:

```
return (StringBuilder) (object) arg;
```

Unboxing conversions can also introduce ambiguities. The following could be an unboxing, numeric, or custom conversion:

int Foo<T> (T x) { return **(int) x**; } // Compile-time error

The solution, again, is to first cast to objectand then to int (which then unambiguously signals an unboxing conversion in this case):

int Foo<T> (T x) { return **(int) (object) x**; }

Covariance

Assuming A is convertible to B, X is covariant if X<A> is convertible to X<B>.

NOTE

With C#'s notion of covariance (and contravariance), "convertible" means convertible via an *implicit reference conversion*—such as A *subclassing* B, or A *implementing* B. Numeric conversions, boxing conversions, and custom conversions are not included. For instance, type IFoo<T> is covariant for T if the following is legal:

IFoo<string> s = ...;
IFoo<object> b = s;

From C# 4.0, generic interfaces permit covariance for (as do generic delegates—see Chapter 4), but generic classes do not. Arrays also support covariance (A[] can be converted to B[] if A has an implicit reference conversion to B), and are discussed here for comparison.

NOTE

Covariance and contravariance (or simply "variance") are advanced concepts. The motivation behind introducing and enhancing variance in C# was to allow generic interface and generic types (in particular, those defined in the Framework, such as IEnumerable<T>) to work *more as you'd expect*. You can benefit from this without understanding the details behind covariance and contravariance.

Classes

Generic classes are not covariant, to ensure **static type safety.**

Consider the following:

```
class Animal {}
class Bear : Animal {}
class Camel : Animal {}

public class Stack<T> // A simple Stack implementation
{
int position;
T[] data = new T[100];
public void Push (T obj) {
data[position++] = obj; }
public T Pop() { return
data[--position]; }
}
```

The following fails to compile:

```
Stack<Bear> bears = new Stack<Bear>();
Stack<Animal> animals = bears; //Compile-time error
```

That restriction prevents the possibility of runtime failure with the following code:

```
animals.Push (new Camel()); // Trying to add Camel to bears
```

Lack of covariance, however, can hinder reusability.

Suppose, for instance, we wanted to write a method to Wash a stack of animals:

```
public class ZooCleaner
{
public static void Wash (Stack<Animal> animals) {...}
}
```

Calling Wash with a stack of bears would generate a compile-time error. One workaround is to redefine the Wash method with a constraint:

```
class ZooCleaner
{
public static void Wash<T> (Stack<T> animals) where T : Animal { ... }
}
```

We can now call Wash as follows:

```
Stack<Bear> bears = new Stack<Bear>();
ZooCleaner.Wash (bears);
```

Another solution is to have Stack<T> implement a covariant generic interface, as we'll see shortly.

Arrays

For historical reasons, array types are covariant. This means that B[] can be cast to A[] if B subclasses A (and both are reference types). For example:

```
Bear[] bears = new Bear[3];
Animal[] animals = bears; // OK
```

The downside of this reusability is that element assignments can fail at runtime:

```
animals[0] = new Camel(); // Runtime error
```

Interfaces

As of C# 4.0, generic interfaces support covariance for type parameters marked with the out modifier. This modifier ensures that, unlike with arrays, covariance with interfaces is fully type-safe. To illustrate, suppose that our Stack class implements the following interface:

```
public interface IPoppable<out T> { T Pop();}
```

The out modifier on T indicates that T is used only in *output positions* (e.g., return types for methods). The out modifier flags the interface as

covariant and allows us to do this:

```
var bears = new Stack<Bear>();
bears.Push (new Bear()); // Bears implements IPoppable<Bear>.We can convert to IPoppable<Animal>:
IPoppable<Animal> animals = bears; // Legal
Animal a = animals.Pop();
```

The cast from bears to animals is permitted by the compiler—by virtue of the interface being covariant. This is type-safe because the case the compiler is trying to avoid—pushing a Camel onto the stack—can't occur as there's no way to feed a Camel *in*to an interface where T can appear only in *out*put positions.

NOTE

Covariance (and contravariance) in interfaces is something that you typically *consume*: it's less common that you need to *write* variant interfaces. Curiously, method parameters marked as out are not eligible for covariance, due to a limitation in the CLR.

We can leverage the ability to cast covariantly to solve the reusability problem described earlier:

```
public class ZooCleaner
{
public static void Wash (IPoppable<Animal>
animals) { ... }
}
```

NOTE

The IEnumerator<T> and IEnumerable<T> interfaces described in Chapter 7 are marked as covariant. This allows you to cast IEnumerable<string> to IEnumerable<object>, for instance.

The compiler will generate an error if you use a covariant type parameter in an *input* position (e.g., a parameter to a method or a writable property).

NOTE

With both generic types and arrays, covariance (and contravariance) is valid only for elements with *reference conversions*—not *boxing*

conversions. So, if you wrote a method that accepted a parameter of type IPoppable<object>, you could call it with IPoppable<string>, but not Ipoppable<int>.

Contravariance

We previously saw that, assuming that A allows an implicit reference conversion to B, a type X is covariant if X<A> allows a reference conversion to X<B>.

A type is *contravariant* when you can convert in the reverse direction—from X<B> to X<A>. This is supported with generic interfaces—when the generic type parameter only appears in *input* positions, designated with the in modifier. Extending our previous example, if the Stack<T> class implements the following interface:

public interface IPushable<**in** T> { void Push (T obj); }

we can legally do this:

IPushable<Animal> animals = new Stack<Animal>();
IPushable<Bear> bears = animals; // Legal
bears.Push (new Bear());

No member in IPushable *outputs* a T, so we can't get into trouble by casting animals to bears (there's no way to Pop, for instance, through that interface).

NOTE

Our Stack<T> class can implement both IPushable<T> and IPoppable<T>—despite T having opposing variance annotations in the two interfaces! This works because you can exercise varianceonly through an interface; therefore, you must commit to the lens of either IPoppable or IPushable before performing a variant conversion. This lens then restricts you to the operations that are legal under the appropriate variance rules. This also illustrates why it would usually make no sense for *classes* (such as

Stack<T>) to be variant: concrete implementations typically require data to flow in both directions.

To give another example, consider the following interface, defined as part of the .NET Framework:

```
public interface IComparer<in T>
{
// Returns a value indicating the relative ordering of a and b

int Compare (T a, T b);

}
```

Because the interface is contravariant, we can use an IComparer<**object**> to compare two *strings*:

```
var objectComparer =Comparer<object>.Default; // objectComparer implements IComparer<object>
IComparer<string> stringComparer =objectComparer;
int result = stringComparer.Compare ("Brett", "Jemaine");
```

Mirroring covariance, the compiler will report an error if you try to use a contravariant parameter in an output position (e.g., as a return value, or in a readable property).

C# Generics Versus C++ Templates

C# generics are similar in application to C++ templates, but they work very differently. In both cases, a synthesis between the producer and consumer must take place, where the placeholder types of the producer are filled in by the consumer. However, with C# generics, producer types (i.e., open types such as List<T>) can be compiled into a library (such as *mscorlib.dll*). This works because the synthesis between the producer and the consumer that produces closed types doesn't actually happen until runtime. With C++ templates, this synthesis is performed at compile time. This means that in C++ you don't deploy template libraries as *.dll*s—they exist only as source code. It also makes it difficult to dynamically inspect, let alone create, parameterized types on the fly.

To dig deeper into why this is the case, consider the Max method in C#, once more:

```
static T Max <T> (T a, T b) where T : IComparable<T>
{
return a.CompareTo (b) > 0 ? a : b;
}
```

Why couldn't we have implemented it like this?

```
static T Max <T> (T a, T b)
{
return a > b ? a : b; //
Compile error
}
```

The reason is that Max needs to be compiled once and work for all possible values of T. Compilation cannot succeed, because there is no single meaning for > across all values of T—in fact, not every T even has a > operator. In contrast, the following code shows the same Max method written with C++ templates. This code will be compiled separately for each value of T, taking on whatever semantics > has for a particular T, failing to compile if a particular T does not support the > operator:

```
template <class T> T Max (T a, T b)
{
return a > b ? a : b;
}
```

The reference type may also be System. ValueType or System.Enum (Chapter 6).