Use the `typeof` operator on a type name.

`GetType` is evaluated at runtime; `typeof` is evaluated statically at compile time.

`System.Type` has properties for such things as the type's name, assembly, base type, and so on. For example:

```
using System;

public class Point { public int X, Y; }

class Test
{
  static void Main()
  {
    Point p = new Point();
    Console.WriteLine (p.GetType().Name);               // Point
    Console.WriteLine (typeof (Point).Name);            // Point
    Console.WriteLine (p.GetType() == typeof(Point)); // True
    Console.WriteLine (p.X.GetType().Name);             // Int32
```

```
    Console.WriteLine (p.GetType() == typeof(Point)); // True
    Console.WriteLine (p.X.GetType().Name);           // Int32
    Console.WriteLine (p.Y.GetType().FullName);       // System.Int32
  }
}
```

`System.Type` also has methods that act as a gateway to the runtime's reflection model, described in Chapter 17.

The ToString Method

The `ToString` method returns the default textual representation of a type instance. This method is overridden by all built-in types. Here is an example of using the `int` type's `ToString` method:

```
int x = 1;
string s = x.ToString();
```

```
string s = x.ToString();
```

```
// s is "1"
```

You can override the ToString method on custom types as follows:

```
public class Panda
{
  public string Name;
  public override string ToString() { return Name; }
}
...
```

```
Panda p = new Panda { Name = "Petey" };
Console.WriteLine (p);    // Petey
```

When you call an *overridden* object member such as ToString

When you call an *overridden* `object` member such as `ToString` directly on a value type, boxing doesn't occur. Boxing then occurs only if you cast:

```
int x = 1;
string s1 = x.ToString();    // Calling on nonboxed value
object box = x;
string s2 = box.ToString();  // Calling on boxed value
```

Object Member Listing

Here are all the members of `object`:

```
public class Object
{
  public Object();
  public extern Type GetType();
```

```
    public extern Type GetType();

public virtual bool Equals (object obj);
public static bool Equals  (object objA, object objB);
public static bool ReferenceEquals (object objA, object objB);

  public virtual int GetHashCode();

  public virtual string ToString();

  protected override void Finalize();
  protected extern object MemberwiseClone();
}
```

We describe the `Equals`, `ReferenceEquals`, and `GetHashCode` methods in “Equality Comparison” on page 245 in Chapter 6.

Structs

A *struct* is similar to a class, with the following key differences:

A struct is a value type, whereas a class is a reference type.

A struct does not support inheritance (other than implicitly deriving from `object`, or more precisely, `System.ValueType`).

A struct can have all the members a class can, except the following:

A parameterless constructor

A finalizer

Virtual members

A struct is used instead of a class when value-type semantics are desirable. Good examples of structs are numeric types, where it is more natural for assignment to copy a value rather than a reference. Because a struct is a value type, each instance does not require instantiation of an object on the heap; this incurs a useful savings when creating many instances of a type. For instance, creating an array of value type requires only a single heap allocation.

Creating

Struct Construction Semantics

The construction semantics of a struct are as follows:

-
-
-

A parameterless constructor that you can't override implicitly exists. This performs a bitwise-zeroing of its fields.

When you define a struct constructor, you must explicitly assign every field.

You can't have field initializers in a struct.

You can't have field initializers in a struct.

Here is an example of declaring and calling struct constructors:

```
public struct Point
{
  int x, y;
  public Point (int x, int y) { this.x = x; this.y = y; }
}
```

```
...
Point p1 = new Point ();
Point p2 = new Point (1, 1);
```

```
// p1.x and p1.y will be 0
// p1.x and p1.y will be 1
```

The next example generates three compile-time errors:

The next example generates three compile-time errors:

```
public struct Point
{
  int x = 1;                        // Illegal: cannot initialize field
  int y;
  public Point() {}                 // Illegal: cannot have
                                    // parameterless constructor
  public Point (int x) {this.x = x;}
}
// Illegal: must assign field y
```


Changing struct to class makes this example legal.

Access Modifiers

Access Modifiers

To promote encapsulation, a type or type member may limit its *accessibility* to other types and other assemblies by adding one of five *access modifiers* to the declaration:

`public`
: Fully accessible; the implicit accessibility for members of an enum or interface

`internal`
: Accessible only within containing assembly or friend assemblies; the default accessibility for non-nested types

`private`
: Visible only within containing type; the default accessibility members of a class or struct

`protected`
: Visible only within containing type or subclasses

`protected internal`
: The *union* of `protected` and `internal` accessibility (this is *less* restrictive than `protected` or `internal` alone)

The CLR has the concept of the *intersection* of protected and

The CLR has the concept of the *intersection* of protected and internal accessibility, but C# does not support this.

Examples

`Class2` is accessible from outside its assembly; `Class1` is not:

```
class Class1 {}                   // Class1 is internal (default)
public class Class2 {}
```

`ClassB` exposes field `x` to other types in the same assembly; `ClassA` does not:

```
class ClassA { int x;          } // x is private (default)
class ClassB { internal int x; }
```

Functions within `Subclass` can call `Bar` but not `Foo`:

```
class BaseClass
```

```
class BaseClass
{
  void Foo()                {}
  protected void Bar() {}
}
class Subclass : BaseClass
{
   void Test1() { Foo(); }
   void Test2() { Bar(); }
}
```

```
// Foo is private (default)
```

```
// Foo is private (default)
// Error - cannot access Foo
// OK
```


Friend Assemblies

In advanced scenarios, you can expose `internal` members to other *friend* assemblies by adding the `System.Runtime.CompilerServices.InternalsVisibleTo` assembly attribute, specifying the name of the friend assembly as follows:

```
[assembly: InternalsVisibleTo ("Friend")]
```

If the friend assembly has a strong name (see Chapter 17), you must specify its *full* 160-byte public key:

```
[assembly: InternalsVisibleTo ("StrongFriend, PublicKey=0024f000048c...")]
```

You can extract the full public key from a strongly named assembly with a LINQ

You can extract the full public key from a strongly named assembly with a LINQ query (we explain LINQ in detail in Chapter 8):

```
string key = string.Join ("",
   Assembly.GetExecutingAssembly().GetName().GetPublicKey()
  .Select (b => b.ToString ("x2"))
  .ToArray());
```

The companion sample in LINQPad invites you to browse to an assembly and then copies the assembly's full public key to the clipboard.

Accessibility Capping

A type caps the accessibility of its declared members. The most common example of capping is when you have an `internal` type with `public` members. For example:

```
class C { public void Foo() {} }
```

`C`'s (default) `internal` accessibility caps `Foo`'s accessibility, effectively making `Foo` `internal`. A common reason `Foo` would be marked `public` is to make for easier refactoring, should `C` later be changed to `public`.

Restrictions on Access Modifiers

When overriding a base class function, accessibility must be identical on the over-

When overriding a base class function, accessibility must be identical on the overridden function. For example:

```
class BaseClass              { protected virtual  void Foo() {} }
class Subclass1 : BaseClass { protected override void Foo() {} }  // OK
class Subclass2 : BaseClass { public    override void Foo() {} }  // Error
```

The compiler prevents any inconsistent use of access modifiers. For example, a subclass itself can be less accessible than a base class, but not more:

```
internal class A {}
public class B : A {}

// Error
```


Interfaces

Interfaces

An interface is similar to a class, but it provides a specification rather than an implementation for its members. An interface is special in the following ways:

- A class can implement *multiple* interfaces. In contrast, a class can inherit from only a *single* class.
- Interface members are *all implicitly abstract*. In contrast, a class can provide both abstract members and concrete members with implementations.
- *Structs* can implement interfaces. In contrast, a struct cannot inherit from a class.

An interface declaration is like a class declaration, but it provides no implementation for its members, since all its members are implicitly abstract. These members will be implemented by the classes and structs that implement the interface. An interface can contain only methods, properties, events, and indexers, which noncoincidentally are precisely the members of a class that can be abstract.

can contain only methods, properties, events, and indexers, which noncoincidentally are precisely the members of a class that can be abstract.

Here is the definition of the `IEnumerator` interface, defined in `System.Collections`:

```
public interface IEnumerator
{
  bool MoveNext();
  object Current { get; }
  void Reset();
}
```

Interface members are always implicitly public and cannot declare an access modifier. Implementing an interface means providing a `public` implementation for all its members:

```
internal class Countdown : IEnumerator
```

```
internal class Countdown : IEnumerator
{
  int count = 11;
  public bool MoveNext () { return count-- > 0 ;               }
  public object Current   { get { return count; }              }
  public void Reset()     { throw new NotSupportedException(); }
}
```

You can implicitly cast an object to any interface that it implements. For example:

```
IEnumerator e = new Countdown();
while (e.MoveNext())
  Console.Write (e.Current);      // 109876543210
```

Even though Countdown is an internal class, its members that implement IEnumerator can be called publicly by casting an instance of Countdown to IEnumerator. For instance, if a public type in the same assembly defined a method as follows:

```
public static class Util
```

```
public static class Util
{
  public static object GetCountDown()
  {
```



```
      return new CountDown();
    }
  }
```

a caller from another assembly could do this:

```
IEnumerator e = (IEnumerator) Util.GetCountDown();
e.MoveNext();
```

If `IEnumerator` was itself defined as `internal`, this wouldn't be possible.

possible.

Extending an Interface

Interfaces may derive from other interfaces. For instance:

```
public interface IUndoable             { void Undo(); }
public interface IRedoable : IUndoable { void Redo(); }
```

`IRedoable` inherits all the members of `IUndoable`.

Explicit Interface Implementation

Implementing multiple interfaces can sometimes result in a collision between member signatures. You can resolve such collisions by *explicitly implementing* an interface member. Consider the following example:

```
interface I1 { void Foo(); }
interface I2 { int Foo(); }
```

```
public class Widget : I1, I2
{
  public void Foo ()
  {
    Console.WriteLine ("Widget's implementation of I1.Foo");
  }
```

```
    Console.WriteLine ("Widget's implementation of I1.Foo");
  }

  int I2.Foo()
  {
    Console.WriteLine ("Widget's implementation of I2.Foo");
    return 42;
  }
}
```

Because both `I1` and `I2` have conflicting `Foo` signatures, `Widget` explicitly implements `I2`'s `Foo` method. This lets the two methods coexist in one class. The only way to call an explicitly implemented member is to cast to its interface:

```
Widget w = new Widget();
w.Foo();                      // Widget's implementation of I1.Foo
((I1)w).Foo();                // Widget's implementation of I1.Foo
((I2)w).Foo();                // Widget's implementation of I2.Foo
```

Another reason to explicitly implement interface members is to hide members that are highly specialized and distracting to a type's normal use case. For example, a type that implements `ISerializable` would typically want to avoid flaunting its `ISerializable` members unless explicitly cast to that interface.

Implementing Interface Members Virtually

An implicitly implemented interface member is, by default, sealed. It must be marked `virtual` or `abstract` in the base class in order to be overridden. For example:

```
public interface IUndoable { void Undo(); }

public class TextBox : IUndoable
{
  public virtual void Undo()
  {
     Console.WriteLine ("TextBox.Undo");
  }
}
```

```
  }
}
public class RichTextBox : TextBox
{
  public override void Undo()
  {
    Console.WriteLine ("RichTextBox.Undo");
  }
}
```

Calling the interface member through either the base class or the interface calls the subclass's implementation:

```
RichTextBox r = new RichTextBox();
r.Undo();                          // RichTextBox.Undo
((IUndoable)r).Undo();             // RichTextBox.Undo
((TextBox)r).Undo();               // RichTextBox.Undo
```

An explicitly implemented interface member cannot be marked virtual, nor can it

An explicitly implemented interface member cannot be marked `virtual`, nor can it be overridden in the usual manner. It can, however, be *reimplemented*.

Reimplementing an Interface in a Subclass

A subclass can reimplement any interface member already implemented by a base class. Reimplementation hijacks a member implementation (when called through the interface) and works whether or not the member is `virtual` in the base class. It also works whether a member is implemented implicitly or explicitly—although it works best in the latter case, as we will demonstrate.

In the following example, `TextBox` implements `IUndoable.Undo` explicitly, and so it cannot be marked as `virtual`. In order to "override" it, `RichTextBox` must reimplement `IUndoable`'s `Undo` method:

```
public interface IUndoable { void Undo(); }
```

```
public class TextBox : IUndoable
{
  void IUndoable.Undo() { Console.WriteLine ("TextBox.Undo"); }
```

```
{
  void IUndoable.Undo() { Console.WriteLine ("TextBox.Undo"); }
}
```



```
public class RichTextBox : TextBox, IUndoable
{
  public new void Undo() { Console.WriteLine ("RichTextBox.Undo"); }
}
```

Calling the reimplemented member through the interface calls the subclass's implementation:

```
RichTextBox r = new RichTextBox();
r.Undo();                 // RichTextBox.Undo      Case 1
((IUndoable)r).Undo();    // RichTextBox.Undo      Case 2
```

Assuming the same RichTextBox definition, suppose that TextBox implemented Undo *implicitly*:

```
public class TextBox : IUndoable
{
```

```
public class TextBox : IUndoable
{
  public void Undo() { Console.WriteLine ("TextBox.Undo"); }
}
```

This would give us another way to call `Undo`, which would "break" the system, as shown in Case 3:


```
RichTextBox r = new RichTextBox();
```

```
RichTextBox r = new RichTextBox();
r.Undo();                          // RichTextBox.Undo
((IUndoable)r).Undo();             // RichTextBox.Undo
((TextBox)r).Undo();               // TextBox.Undo
```

Case 1
Case 2
Case 3

Case 3 shows that reimplementation hijacking is effective only when a member is called through the interface and not through the base class. This is usually undesirable, as it can mean inconsistent semantics. Reimplementation is most appropriate as a strategy for overriding *explicitly* implemented interface members.

Alternatives to interface reimplementation

Even with explicit member implementation, interface reimplementation is prob

Even with explicit member implementation, interface reimplementation is problematic for a couple of reasons:

- The subclass has no way to call the base class method.
- The base class author may not anticipate that a method be reimplemented and may not allow for the potential consequences.

Reimplementation can be a good last resort when subclassing hasn't been anticipated. A better option, however, is to design a base class such that reimplementation will never be required. There are two ways to achieve this:

- When implicitly implementing a member, mark it `virtual` if appropriate.
- When explicitly implementing a member, use the following pattern if you anticipate that subclasses might need to override any logic:

ticipate that subclasses might need to override any logic:

```
public class TextBox : IUndoable
{
  void IUndoable.Undo()          { Undo(); }   // Calls method below
  protected virtual void Undo() { Console.WriteLine ("TextBox.Undo"); }
}
```



```
public class RichTextBox : TextBox
{
  protected override void Undo() { Console.WriteLine("RichTextBox.Undo"); }
}
```

If you don't anticipate any subclassing, you can mark the class as `sealed` to preempt interface reimplementation.

Interfaces and Boxing

Casting a struct to an interface causes boxing. Calling an implicitly implemented member on a struct does not cause boxing:

```
interface  I { void Foo();          }
struct S : I { public void Foo() {} }
```

```
...
S s = new S();
s.Foo();
I i = s;
i.Foo();
```

```
// No boxing.
// Box occurs when casting to interface.
```

Writing a Class Versus an Interface

Writing a Class Versus an Interface

As a guideline:

- Use classes and subclasses for types that naturally share an implementation.
- Use interfaces for types that have independent implementations.

Consider the following classes:

```
abstract class Animal {}
abstract class Bird           : Animal {}
abstract class Insect         : Animal {}
abstract class FlyingCreature : Animal {}
abstract class Carnivore      : Animal {}

// Concrete classes:

class Ostrich : Bird {}
class Eagle   : Bird, FlyingCreature, Carnivore {}  // Illegal
class Bee     : Insect, FlyingCreature {}           // Illegal
class Flea    : Insect, Carnivore {}                // Illegal
```

The `Eagle`, `Bee`, and `Flea` classes do not compile because inheriting from multiple classes is prohibited. To resolve this, we must convert some of the types to interfaces. The question then arises, which types? Following our general rule, we could say that insects share an implementation, and birds share an implementation, so faces. The question then arises, which types? Following our general rule, we could say that insects share an implementation, and birds share an implementation, so they remain classes. In contrast, flying creatures have independent mechanisms for flying, and carnivores have independent strategies for eating animals, so we would convert `FlyingCreature` and `Carnivore` to interfaces:

```
interface IFlyingCreature {}
interface ICarnivore      {}
```


In a typical scenario, `Bird` and `Insect` might correspond to a Windows control and a web control; `FlyingCreature` and `Carnivore` might correspond to `IPrintable` and `IUndoable`.

Enums

An enum is a special value type that lets you specify a group of named numeric constants. For example:

```
public enum BorderSide { Left, Right, Top, Bottom }
```

```
public enum BorderSide { Left, Right, Top, Bottom }
```

We can use this enum type as follows:

```
BorderSide topSide = BorderSide.Top;
bool isTop = (topSide == BorderSide.Top);
```

```
// true
```

Each enum member has an underlying integral value. By default:

-
-

Underlying values are of type `int`.

The constants `0`, `1`, `2`... are automatically assigned, in the declaration order of the enum members.

You may specify an alternative integral type, as follows:

```
public enum BorderSide : byte { Left, Right, Top, Bottom }
```

You may also specify an explicit underlying value for each enum member:

```
public enum BorderSide : byte { Left=1, Right=2, Top=10, Bottom=11 }
```

The compiler also lets you explicitly assign *some* of the enum members. The unassigned enum members keep incrementing from the last explicit value. The preceding example is equivalent to the following:

```
public enum BorderSide : byte
  { Left=1, Right, Top=10, Bottom }
```

Enum Conversions

You can convert an `enum` instance to and from its underlying integral value with an explicit cast:

```
int i = (int) BorderSide.Left;
BorderSide side = (BorderSide) i;
bool leftOrRight = (int) side <= 2;
```

You can also explicitly cast one enum type to another. Suppose `HorizontalAlign`

You can also explicitly cast one enum type to another. Suppose HorizontalAlignment is defined as follows:

```
public enum HorizontalAlignment
{
  Left = BorderSide.Left,
  Right = BorderSide.Right,
```



```
  Center
}
```

A translation between the enum types uses the underlying integral values:

```
HorizontalAlignment h = (HorizontalAlignment) BorderSide.Right;
// same as:
```

```
HorizontalAlignment h = (HorizontalAlignment) BorderSide.Right;
// same as:
HorizontalAlignment h = (HorizontalAlignment) (int) BorderSide.Right;
```

The numeric literal `0` is treated specially by the compiler in an `enum` expression and does not require an explicit cast:

```
BorderSide b = 0;    // No cast required
if (b == 0) ...
```

There are two reasons for the special treatment of `0`:

The first member of an enum is often used as the "default" value.

For *combined enum* types, `0` means "no flags."

Flags Enums

Flags Enums

You can combine enum members. To prevent ambiguities, members of a combinable enum require explicitly assigned values, typically in powers of two. For example:

```
[Flags]
public enum BorderSides { Left=1, Right=2, Top=4, Bottom=8 }
```

To work with combined enum values, you use bitwise operators, such as | and &. These operate on the underlying integral values:

```
BorderSides leftRight = BorderSides.Left | BorderSides.Right;
```

```
if ((leftRight & BorderSides.Left) != 0)
  Console.WriteLine ("Includes Left");
string formatted = leftRight.ToString();
```

```
// Includes Left
// "Left, Right"
```

```
// "Left, Right"
```

```
BorderSides s = BorderSides.Left;
s |= BorderSides.Right;
Console.WriteLine (s == leftRight);
s ^= BorderSides.Right;
Console.WriteLine (s);
```

```
// True
// Toggles BorderSides.Right
// Left
```

By convention, the `Flags` attribute should always be applied to an enum type when its members are combinable. If you declare such an `enum` without the `Flags` attribute

By convention, the `Flags` attribute should always be applied to an enum type when its members are combinable. If you declare such an `enum` without the `Flags` attribute, you can still combine members, but calling `ToString` on an `enum` instance will emit a number rather than a series of names.

By convention, a combinable enum type is given a plural rather than singular name.

For convenience, you can include combination members within an enum declaration itself:


```
[Flags]
public enum BorderSides
{
  Left=1, Right=2, Top=4, Bottom=8,
  LeftRight = Left | Right,
  TopBottom = Top  | Bottom,
```

```
  TopBottom = Top       | Bottom,
  All       = LeftRight | TopBottom
}
```

Enum Operators

The operators that work with enums are:

```
=    ==    !=    <    >    <=    >=    +    -
+=    -=    ++   -    sizeof
```

```
^
```

```
&
```

```
|
```

~

The bitwise, arithmetic, and comparison operators return the result of processing the underlying integral values. Addition is permitted between an enum and an integral type, but not between two enums.

Type-Safety Issues

Consider the following enum:

```
public enum BorderSide { Left, Right, Top, Bottom }
```

Since an enum can be cast to and from its underlying integral type, the actual value it may have may fall outside the bounds of a legal enum member. For example:

```
BorderSide b = (BorderSide) 12345;
Console.WriteLine (b);                    // 12345
```

The bitwise and arithmetic operators can produce similarly invalid values:

```
BorderSide b = BorderSide.Bottom;
b++;
// No errors
```

```
// No errors
```

An invalid `BorderSide` would break the following code:

```
void Draw (BorderSide side)
{
  if      (side == BorderSide.Left)  {...}
  else if (side == BorderSide.Right) {...}
  else if (side == BorderSide.Top)   {...}
  else                               {...} // Assume BorderSide.Bottom
}
```

One solution is to add another `else` clause:

```
...
else if (side == BorderSide.Bottom) ...
else throw new ArgumentException ("Invalid BorderSide: " + side, "side");
```

Another workaround is to explicitly check an enum value for validity. The static `Enum.IsDefined` method does this job:

```
BorderSide side = (BorderSide) 12345;
Console.WriteLine (Enum.IsDefined (typeof (BorderSide), side));   // False
```

```
Console.WriteLine (Enum.IsDefined (typeof (BorderSide), side));   // False
```


Unfortunately, `Enum.IsDefined` does not work for flagged enums. However, the following helper method (a trick dependent on the behavior of `Enum.ToString()`) returns `true` if a given flagged enum is valid:

```
static bool IsFlagDefined (Enum e)
{
  decimal d;
  return !decimal.TryParse(e.ToString(), out d);
}
```

```
[Flags]
public enum BorderSides { Left=1, Right=2, Top=4, Bottom=8 }

static void Main()
{
  for (int i = 0; i <= 16; i++)
  {
    BorderSides side = (BorderSides)i;
```

```
    BorderSides side = (BorderSides)i;
    Console.WriteLine (IsFlagDefined (side) + " " + side);
  }
}
```

Nested Types

A *nested type* is declared within the scope of another type. For example:

```
public class TopLevel
{
  public class Nested { }
  public enum Color { Red, Blue, Tan }
}
```

```
// Nested class
// Nested enum
```

A nested type has the following features:

- It can access the enclosing type's private members and everything else the enclosing type can access.
- It can be declared with the full range of access modifiers, rather than just `public` and `internal`.
- The default visibility for a nested type is `private` rather than `internal`.
- Accessing a nested type from outside the enclosing type requires qualification with the enclosing type's name (like when accessing static members).

Accessing a nested type from outside the enclosing type requires qualification with the enclosing type's name (like when accessing static members).

For example, to access `Color.Red` from outside our `TopLevel` class, we'd have to do this:

```
TopLevel.Color color = TopLevel.Color.Red;
```

All types can be nested; however, only classes and structs can *nest*.

Here is an example of accessing a private member of a type from a nested type:


```
public class TopLevel
{
  static int x;
  class Nested
  {
    static void Foo() { Console.WriteLine (TopLevel.x); }
  }
}
```

```
  }
}
```

Here is an example of applying the `protected` access modifier to a nested type:

```
public class TopLevel
{
  protected class Nested { }
}
```

```
public class SubTopLevel : TopLevel
{
  static void Foo() { new TopLevel.Nested(); }
}
```

Here is an example of referring to a nested type from outside the enclosing type:

Here is an example of referring to a nested type from outside the enclosing type:


```
public class TopLevel
{
  public class Nested { }
}
```

```
  public class Nested { }
}

class Test
{
  TopLevel.Nested n;
}
```

Nested types are used heavily by the compiler itself when it generates private classes that capture state for constructs such as iterators and anonymous methods.

If the sole reason for using a nested type is to avoid cluttering a namespace with too many types, consider using a nested namespace instead. A nested type should be used because of its stronger access control restrictions, or when the nested class must access private members of the containing class.

access private members of the containing class.

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*.

C# generics and C++ templates are similar concepts, but they work differently. We explain this difference in "C# Generics Versus C++ Templates" on page 113.

Generic Types

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)
  public T Pop()
}
```

```
}
```

```
{ data[position++] = obj;  }
{ 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 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)
  public int Pop()
}
```

```
{ data[position++] = obj;  }
{ return data[--position]; }
```

```
{ 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
{
```

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

Creating Types

[redacted]

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");
int i = (int)stack.Pop();
```

```
// Wrong type, but no error!
// 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 a specific 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 *gen-*

lowing 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.

`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 general-purpose way only. Here is a generic method that swaps two values of any type:

```
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>() { }
```

```
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; set; }
}
```

```
}
```

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:

conflict:

```
class A<T> {}
class A<T1,T2> {}
```

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<>);
Type a2 = typeof (A<,>);
```

```
Type a2 = typeof (A<,>);
```

```
// Unbound type (notice no type arguments).
// Use commas to indicate multiple type args.
```

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)
```

```
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 SomeClass and

In the following example, `GenericClass<T,U>` requires `T` to derive from `SomeClass` and implement `Interface1`, and requires `U` to provide a parameterless constructor:

```
class     SomeClass {}
interface Interface1 {}
```

```
class GenericClass<T> 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* or *interface constraint* specifies that the type parameter must subclass or implement a particular class or interface. This allows instances of that type to be implicitly cast 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);
}
```

```
  int CompareTo (T other);
}
```

`CompareTo` returns a positive number if `other` is greater than `this`. 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 the section "Nullable Types" on page 148 in Chapter 4):

Types" on page 148 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 *another type parameter*. In this example, the method `FilteredStack` returns another `Stack`, containing only the subset of elements where the type parameter `T` is of the type parameter `U`:

```
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 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> {...}
```

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> {...}
```

```
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;
  }
```

```
      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);
```

```
    {
      Console.WriteLine (++Bob<int>.Count);
      Console.WriteLine (++Bob<int>.Count);
      Console.WriteLine (++Bob<string>.Count);
      Console.WriteLine (++Bob<object>.Count);
    }
}
```

```
// 1
// 2
// 1
// 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 creates an 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.

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;
```

```
    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 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 `object` and then to `int` (which then unambiguously signals an unboxing conversion in this case):

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

Covariance

Assuming `S` subclasses `B`, type `X` is *covariant* if `X<S>` allows a reference conversion to `X<B>`.

In other words, type `IFoo<T>` is covariant if the following is legal:

In other words, type IFoo<T> is covariant if the following is legal:

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

As of C# 4.0, generic interfaces permit covariance (as do generic delegates—see Chapter 4), but generic classes do not. Arrays also support covariance (`S[]` can be cast to `B[]` if `S` subclasses `B`), and are discussed here for comparison.

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.

variance 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<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:

around 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(); }
```

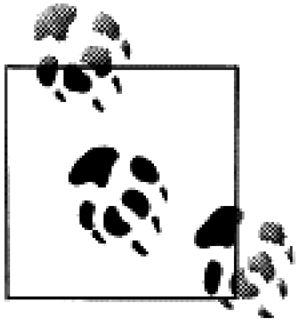
The `out` modifier on `T` is new to C# 4.0 and indicates that `T` is used only in *output positions* (e.g., return types for methods). The `out` modifier flags the interface as *covariant* and allows us to do this:

positions (e.g., return types for methods). The out modifier flags the interface as *covariant* and allows us to do this:

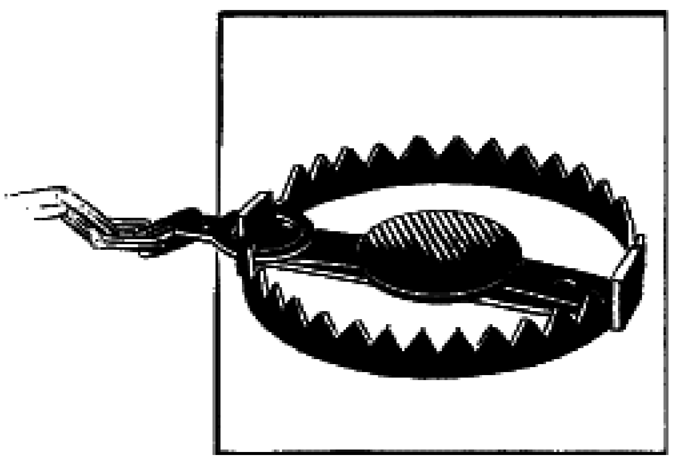

```
var bears = new Stack<Bear>();
bears.Push (new Bear());
// Bears implements IPoppable<Bear>. We can convert to IPoppable<Animal>:
IPoppable<Animal> animals = bears;   // Legal
Animal a = animals.Pop();
```

The cast from bears to animals is permitted by the compiler—by virtue of the in-

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.



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.

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) { ... }
}
```


The `IEnumerator<T>` and `IEnumerable<T>` interfaces described in Chapter 7 are marked as covariant from Framework 4.0. 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).

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 a type X is covariant if X<S> allows a reference conversion to X<B> where S subclasses B. A type is *contravariant* when you can convert in the reverse direction—from X<B> to X<S>. This is supported in C# 4.0 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>();
```

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).

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 variance only 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 make no sense for *classes* (such as `Stack<T>`) to be variant.

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

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

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 *.dlls*—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
```

```
    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;
}
```

4

Advanced C#

In this chapter, we cover advanced C# topics that build on concepts explored in previous chapters. You should read the first four sections sequentially; you can read the remaining sections in any order.

Delegates

A delegate dynamically wires up a method caller to its target method. There are two aspects to a delegate: *type* and *instance*. A *delegate type* defines a *protocol* to which

A delegate dynamically wires up a method caller to its target method. There are two aspects to a delegate: *type* and *instance*. A *delegate type* defines a *protocol* to which the caller and target will conform, comprising a list of parameter types and a return type. A *delegate instance* is an object that refers to one (or more) target methods conforming to that protocol.

A delegate instance literally acts as a delegate for the caller: the caller invokes the delegate, and then the delegate calls the target method. This indirection decouples the caller from the target method.

A delegate type declaration is preceded by the keyword **delegate**, but otherwise it resembles an (abstract) method declaration. For example:

```
delegate int Transformer (int x);
```

To create a delegate instance, you can assign a method to a delegate variable:

```
class Test
{
  static void Main()
  {
    Transformer t = Square;          // Create delegate instance
    int result = t(3);               // Invoke delegate
    Console.WriteLine (result);      // 9
```

```
    int result = t(3);                    // Invoke delegate
    Console.WriteLine (result);           // 9
  }
  static int Square (int x) { return x * x; }
}
```

115

Technically, we are specifying a *method group* when we refer to `Square` without brackets or arguments. If the method is overloaded, C# will pick the correct overload based on the signature of the delegate to which it's being assigned.

Invoking a delegate is just like invoking a method (since the delegate's purpose is merely to provide a level of indirection):

```
t(3);
```

The statement:

```
Transformer t = Square;
```