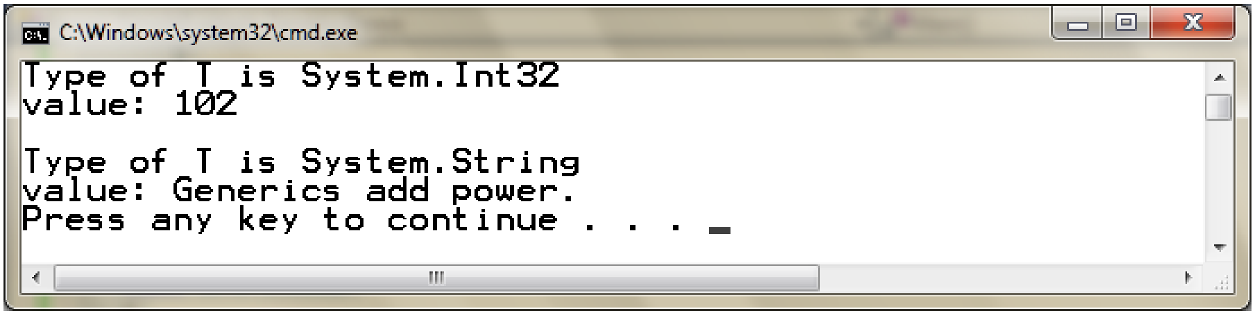
Generic

Thí dụ đơn giản về Generic

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace GenericSimpleExample
{
    // Khai bao lớp Generic: Gen.
    // T trong lớp Gen là tham số kiểu (type parameter)
    // T sẽ được thay thế với một kiểu dữ liệu cụ thể khi đối tượng lớp được tạo.
    class Gen<T>
    {
        T ob; // khai báo biến kiểu T
        // Hàm tạo có tham số kiểu T.
        public Gen(T o)
        {
            ob = o;
        }
        // trả về ob có kiểu T.
        public T GetOb()
        {
            return ob;
        }
        // Cho biết kiểu T.
        public void ShowType()
        {
            Console.WriteLine("Type of T is " + typeof(T));
        }
    }
    // Sử dụng lớp Generic.
    class GenericsDemo
    {
        static void Main()
        {
            // Tạo một thể hiện lớp Gen với kiểu T là số nguyên.
            Gen<int> iOb;
            iOb = new Gen<int>(102);
            iOb.ShowType();
            // Get the value in iOb.
            int v = iOb.GetOb();
            Console.WriteLine("value: " + v);
            Console.WriteLine();
            // Tạo một thể hiện lớp Gen với kiểu T là strings.
            Gen<string> strOb = new Gen<string>("Generics add power.");
            strOb.ShowType();
            // Lấy giá trị của strOb.
            string str = strOb.GetOb();
            Console.WriteLine("value: " + str);
        }
    }
}
```

Kết quả:

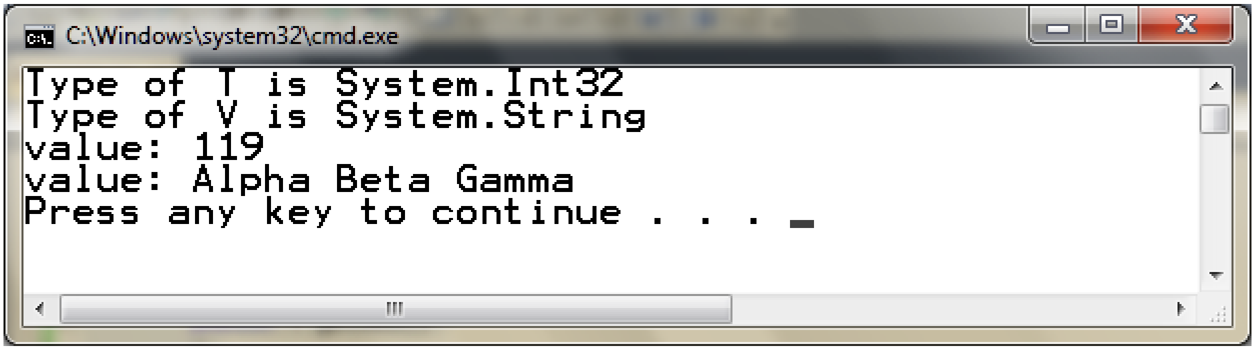


```
Type of T is System.Int32
value: 102

Type of T is System.String
value: Generics add power.
Press any key to continue . . . _
```

A Generic Class with Two Type Parameters

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace GenericCLassWithTwoTypeParameters
{
    // Khai bao lớp Generic: Gen.
    // T và V  trong lớp TwoGen là hai tham số kiểu (type parameter)
    // T và V  sẽ được thay thế với một kiểu dữ liệu cụ thể khi đối tượng lớp được tạo.

    class TwoGen<T, V>
    {
              T ob1; // khai báo biến kiểu  T
              V ob2; // khai báo biến kiểu V
        // Hàm tạo có hai tham số kiểu T và V.
        public TwoGen(T o1, V o2)
        {
            ob1 = o1;
            ob2 = o2;
        }
              // // Cho biết kiểu T và V.
        public void showTypes()
        {
            Console.WriteLine("Type of T is " + typeof(T));
            Console.WriteLine("Type of V is " + typeof(V));
        }
        public T getob1()
        {
            return ob1;
        }
        public V GetObj2()
        {
            return ob2;
        }
    }
  // Sử dụng lớp Generic.
    class SimpGen
    {
        static void Main()
        {
            TwoGen<int, string> tgObj =
            new TwoGen<int, string>(119, "Alpha Beta Gamma");

            tgObj.showTypes();

            int v = tgObj.getob1();
            Console.WriteLine("value: " + v);

             string str = tgObj.GetObj2();
            Console.WriteLine("value: " + str);
        }
    }
}
```

Kết quả:



```
Type of T is System.Int32
Type of V is System.String
value: 119
value: Alpha Beta Gamma
Press any key to continue . . .
```

Dạng tổng quát một lớp Generic

The generics syntax shown in the preceding examples can be generalized. Here is the syntax for declaring a generic class:

class *class-name<type-param-list>* { // ...

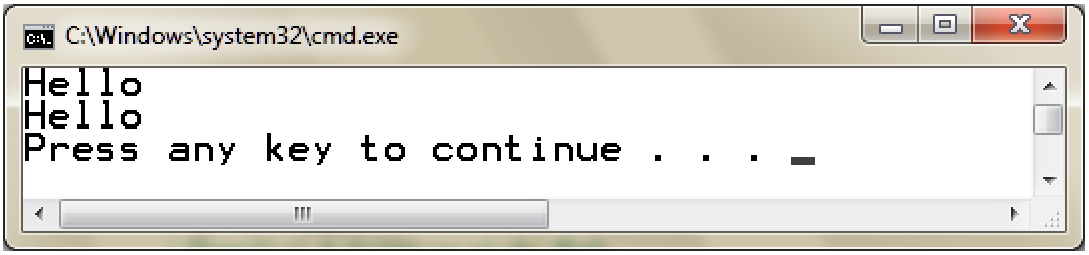
Cú pháp để khai báo một thể hiện của lớp generics:
***class-name*<*type-arg-list*> *var-name* = new *class-name*<*type-arg-list*>(*cons-arg-list*);**

Ràng buộc tham số kiểu
Cú pháp:
class *class-name*<*type-param*> where *type-param* : *constraints* { // ...

Thí dụ về tham số kiểu có ràng buộc lớp cơ sở

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SimpleBaseClassConstraint
{
    // A simple demonstration of a base class constraint.
    class A
    {
        public void Hello()
        {
            Console.WriteLine("Hello");
        }
    }
    // Lớp B dẫn xuất từ lớp A.
    class B : A { }
    // Lớp C không dẫn xuất từ lớp A.
    class C { }
    // Khai báo lớp Test có tham số kiểu T có ràng buộc kiểu lớp cơ sở A
    class Test<T> where T : A
    {
        T obj;
        public Test(T o)
        {
            obj = o;
        }
        public void SayHello()
        {
            // gọi phương thức Hello() được khai báo trong lớp cơ sở A.
            obj.Hello();
        }
    }
    class BaseClassConstraintDemo
    {
        static void Main()
        {
            A a = new A();
            B b = new B();
            C c = new C();
            // Hợp lệ vì A là lớp cơ sở chỉ định.
            Test<A> t1 = new Test<A>(a);
            t1.SayHello();
            // Hợp lệ vì B là lớp dẫn xuất từ A.
            Test<B> t2 = new Test<B>(b);
            t2.SayHello();
            // Không hợp lệ vì C không được dẫn xuất từ A.
            // Test<C> t3 = new Test<C>(c); // Error!
            // t3.SayHello(); // Error!
        }
    }
}
```

Kết quả:



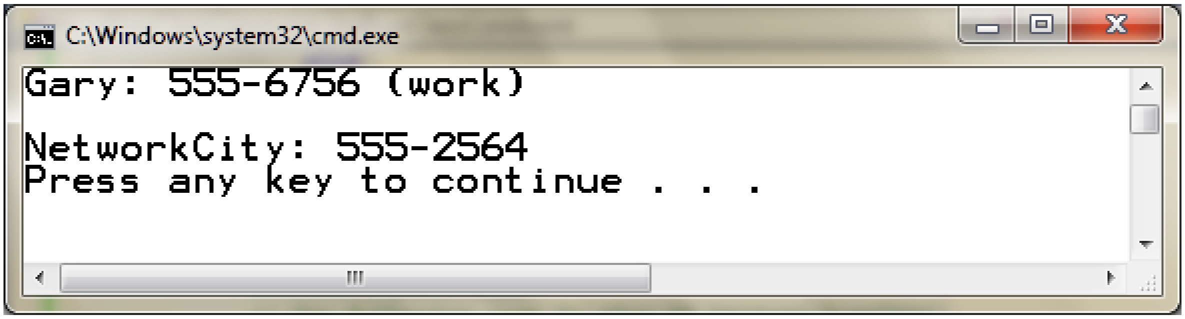
```
Hello
Hello
Press any key to continue . . . _
```

Thêm một thí dụ nữa về tham số kiểu có ràng buộc lớp cơ sở

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace MoreBaseClassConstraint
{
    // A more practical demonstration of a base class constraint.
    // A custom exception that is thrown if a name or number is not found.
    class NotFoundException : Exception
    {
        /* Implement all of the Exception constructors. Notice that
        the constructors simply execute the base class constructor.
        Because NotFoundException adds nothing to Exception,
        there is no need for any further actions. */
        public NotFoundException() : base() { }
        public NotFoundException(string str) : base(str) { }
        public NotFoundException(string str, Exception inner) :
```

```
        base(str, inner) { }
    protected NotFoundException(
    System.Runtime.Serialization.SerializationInfo si,
    System.Runtime.Serialization.StreamingContext sc) :
        base(si, sc) { }
}
// A base class that stores a name and phone number.
class PhoneNumber
{
    public PhoneNumber(string n, string num)
    {
        Name = n;
        Number = num;
    }
    public string Number { get; set; }
    public string Name { get; set; }
}
// A class of phone numbers for friends.
class Friend : PhoneNumber
{
    public Friend(string n, string num, bool wk) :
        base(n, num)
    {
        IsWorkNumber = wk;
    }
    public bool IsWorkNumber { get; private set; }
    // ...
}
// A class of phone numbers for suppliers.
class Supplier : PhoneNumber
{
    public Supplier(string n, string num) :
        base(n, num) { }
    // ...
}
// Notice that this class does not inherit PhoneNumber.
class EmailFriend
{
    // ...
}
// PhoneList can manage any type of phone list
// as long as it is derived from PhoneNumber.
class PhoneList<T> where T : PhoneNumber
{
    T[] phList;
    int end;
    public PhoneList()
    {
        phList = new T[10];
        end = 0;
    }
    // Add an entry to the list.
    public bool Add(T newEntry)
    {
        if (end == 10) return false;
        phList[end] = newEntry;
        end++;
        return true;
    }
    // Given a name, find and return the phone info.
    public T FindByName(string name)
    {
        for (int i = 0; i < end; i++)
        {
            // Name can be used because it is a member of
            // PhoneNumber, which is the base class constraint.
            if (phList[i].Name == name)
                return phList[i];
        }
        // Name not in list.
        throw new NotFoundException();
    }
    // Given a number, find and return the phone info.
    public T FindByNumber(string number)
    {
        for (int i = 0; i < end; i++)
        {
            // Number can be used because it is also a member of
            // PhoneNumber, which is the base class constraint.
            if (phList[i].Number == number)
                return phList[i];
        }
        // Number not in list.
        throw new NotFoundException();
    }
    // ...
}
// Demonstrate base class constraints.
class UseBaseClassConstraint
{
    static void Main()
    {
            // The following code is OK because Friend
            // inherits PhoneNumber.
            PhoneList<Friend> plist = new PhoneList<Friend>();
            plist.Add(new Friend("Tom", "555-1234", true));
            plist.Add(new Friend("Gary", "555-6756", true));
            plist.Add(new Friend("Matt", "555-9254", false));
            try
            {
                // Find the number of a friend given a name.
                Friend frnd = plist.FindByName("Gary");
                Console.Write(frnd.Name + ": " + frnd.Number);
                if (frnd.IsWorkNumber)
                    Console.WriteLine(" (work)");
                else
                    Console.WriteLine();
            }
            catch (NotFoundException)
            {
                Console.WriteLine("Not Found");
            }
            Console.WriteLine();
            // The following code is also OK because Supplier
            // inherits PhoneNumber.
            PhoneList<Supplier> plist2 = new PhoneList<Supplier>();
            plist2.Add(new Supplier("Global Hardware", "555-8834"));
            plist2.Add(new Supplier("Computer Warehouse", "555-9256"));
            plist2.Add(new Supplier("NetworkCity", "555-2564"));
            try
            {
                // Find the name of a supplier given a number.
                Supplier sp = plist2.FindByNumber("555-2564");
                Console.WriteLine(sp.Name + ": " + sp.Number);
            }
            catch (NotFoundException)
            {
                Console.WriteLine("Not Found");
            }
            // The following declaration is invalid because EmailFriend
            // does NOT inherit PhoneNumber.
            // PhoneList<EmailFriend> plist3 =
            // new PhoneList<EmailFriend>(); // Error!
        }
    }
}
```

Kết quả:



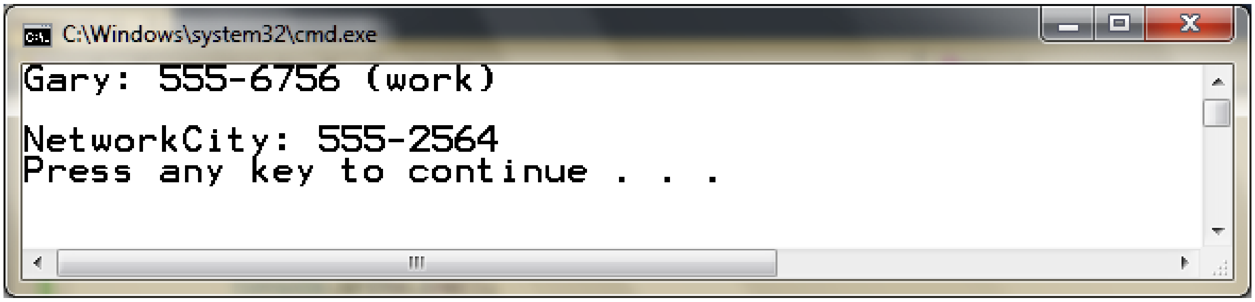
```
Gary: 555-6756 (work)

NetworkCity: 555-2564
Press any key to continue . . .
```

Using an Interface Constraint

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace InterfaceConstraint
{
    // Use an interface constraint.
    // A custom exception that is thrown if a name or number is not found.
    class NotFoundException : Exception
    {
        /* Implement all of the Exception constructors. Notice that
         the constructors simply execute the base class constructor.
        Because NotFoundException adds nothing to Exception,
        there is no need for any further actions. */
        public NotFoundException() : base() { }
        public NotFoundException(string str) : base(str) { }
        public NotFoundException(string str, Exception inner) :
            base(str, inner) { }
        protected NotFoundException(
        System.Runtime.Serialization.SerializationInfo si,
        System.Runtime.Serialization.StreamingContext sc) :
            base(si, sc) { }
    }
    // An interface that supports a name and phone number.
    public interface IPhoneNumber
    {
        string Number
        {
            get;
            set;
        }
        string Name
        {
            get;
            set;
        }
    }
    // A class of phone numbers for friends.
    // It implements IPhoneNumber.
    class Friend : IPhoneNumber
    {
        public Friend(string n, string num, bool wk)
        {
            Name = n;
            Number = num;
            IsWorkNumber = wk;
        }
        public bool IsWorkNumber { get; private set; }
        // Implement IPhoneNumber.
        public string Number { get; set; }
        public string Name { get; set; }
        // ...
    }
    // A class of phone numbers for suppliers.
    class Supplier : IPhoneNumber
    {
        public Supplier(string n, string num)
        {
            Name = n;
            Number = num;
        }
        // Implement IPhoneNumber.
        public string Number { get; set; }
        public string Name { get; set; }
        // ...
    }
    // Notice that this class does not implement IPhoneNumber.
    class EmailFriend
    {
        // ...
    }
    // PhoneList can manage any type of phone list
    // as long as it implements IPhoneNumber.
    class PhoneList<T> where T : IPhoneNumber
    {
        T[] phList;
        int end;
        public PhoneList()
        {
            phList = new T[10];
            end = 0;
        }
        public bool Add(T newEntry)
        {
            if (end == 10) return false;
            phList[end] = newEntry;
            end++;
            return true;
        }
        // Given a name, find and return the phone info.
        public T FindByName(string name)
        {
            for (int i = 0; i < end; i++)
            {
                // Name can be used because it is a member of
                // IPhoneNumber, which is the interface constraint.
                if (phList[i].Name == name)
                    return phList[i];
            }
            // Name not in list.
            throw new NotFoundException();
        }
        // Given a number, find and return the phone info.
        public T FindByNumber(string number)
        {
            for (int i = 0; i < end; i++)
            {
                // Number can be used because it is also a member of
                // IPhoneNumber, which is the interface constraint.
                if (phList[i].Number == number)
                    return phList[i];
            }
            // Number not in list.
            throw new NotFoundException();
        }
        // ...
    }
    // Demonstrate interface constraints.
    class UseInterfaceConstraint
    {
        static void Main()
        {
            // The following code is OK because Friend
            // implements IPhoneNumber.
            PhoneList<Friend> plist = new PhoneList<Friend>();
            plist.Add(new Friend("Tom", "555-1234", true));
```

```
            plist.Add(new Friend("Gary", "555-6756", true));
            plist.Add(new Friend("Matt", "555-9254", false));
            try
            {
                // Find the number of a friend given a name.
                Friend frnd = plist.FindByName("Gary");
                Console.Write(frnd.Name + ": " + frnd.Number);
                if (frnd.IsWorkNumber)
                    Console.WriteLine(" (work)");
                else
                    Console.WriteLine();
            }
            catch (NotFoundException)
            {
                Console.WriteLine("Not Found");
            }
            Console.WriteLine();
            // The following code is also OK because Supplier
            // implements IPhoneNumber.
            PhoneList<Supplier> plist2 = new PhoneList<Supplier>();
            plist2.Add(new Supplier("Global Hardware", "555-8834"));
            plist2.Add(new Supplier("Computer Warehouse", "555-9256"));
            plist2.Add(new Supplier("NetworkCity", "555-2564"));
            try
            {
                // Find the name of a supplier given a number.
                Supplier sp = plist2.FindByNumber("555-2564");
                Console.WriteLine(sp.Name + ": " + sp.Number);
            }
            catch (NotFoundException)
            {
                Console.WriteLine("Not Found");
            }
            // The following declaration is invalid because EmailFriend
            // does NOT implement IPhoneNumber.
            // PhoneList<EmailFriend> plist3 =
            // new PhoneList<EmailFriend>(); // Error!
        }
    }

}
```

Kết quả:



```
Gary: 555-6756 (work)

NetworkCity: 555-2564
Press any key to continue . . .
```

Using the new( ) Constructor Constraint

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace InterfaceConstraint
{
   // Demonstrate a new() constructor constraint.
   class MyClass
   {
      public MyClass()
      {
         // ...
      }
      //...
   }
   class Test<T> where T : new()
   {
      T obj;
      public Test()
      {
         // This works because of the new() constraint.
         obj = new T(); // create a T object
      }
      // ...
   }
   class ConsConstraintDemo
   {
      static void Main()
      {
         Test<MyClass> x = new Test<MyClass>();
      }
   }
}
```

The Reference Type and Value Type Constraints

Với class

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace InterfaceConstraint
{
    // Demonstrate a reference constraint.

    class MyClass
    {
        //...
    }
    // Use a reference constraint.
    class Test<T> where T : class
    {
        T obj;
        public Test()
        {
            // The following statement is legal only
            // because T is guaranteed to be a reference
            // type, which can be assigned the value null.
            obj = null;
        }
        // ...
    }
    class ClassConstraintDemo
    {
        static void Main()
        {
            // The following is OK because MyClass is a class.
            Test<MyClass> x = new Test<MyClass>();
            // The next line is in error because int is a value type.
            // Test<int> y = new Test<int>();
        }
    }
}
```

Với Struct

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace InterfaceConstraint
{
    // Demonstrate a value type constraint.
    struct MyStruct
    {
        //...
    }
    class MyClass
    {
        // ...
    }
    class Test<T> where T : struct
    {
        T obj;
        public Test(T x)
        {
            obj = x;
        }
        // ...
    }
    class ValueConstraintDemo
    {
        static void Main()
        {
            // Both of these declarations are legal.
            Test<MyStruct> x = new Test<MyStruct>(new MyStruct());
            Test<int> y = new Test<int>(10);
            // But, the following declaration is illegal!
            // Test<MyClass> z = new Test<MyClass>(new MyClass());
        }
    }
}
```

Using a Constraint to Establish a Relationship Between Two Type Parameters

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace InterfaceConstraint
{
    // Create relationship between two type parameters.
    using System;
    class A
    {
        //...
    }
    class B : A
    {
        // ...
    }
    // Here, V must be or inherit from T.
    class Gen<T, V> where V : T
    {
        // ...
    }
    class NakedConstraintDemo
    {
        static void Main()
        {
            // This declaration is OK because B inherits A.
            Gen<A, B> x = new Gen<A, B>();
            // This declaration is in error because
            // A does not inherit B.
            // Gen<B, A> y = new Gen<B, A>();
        }
    }
}
```

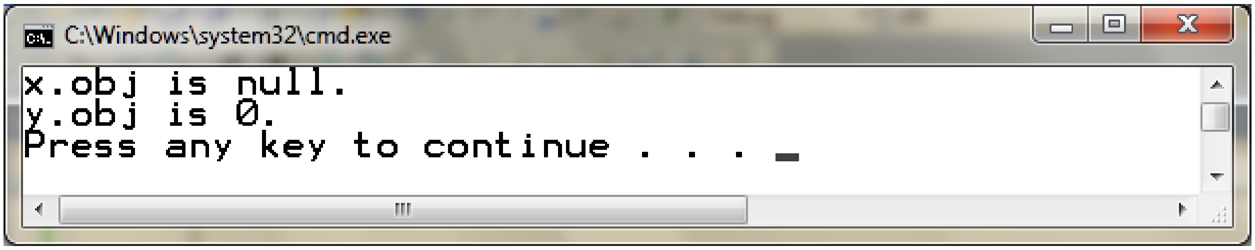
Using Multiple Constraints

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace InterfaceConstraint
{
    // Use multiple where clauses.
    // Gen has two type arguments and both have a where clause.
    class Gen<T, V>
        where T : class
        where V : struct
    {
        T ob1;
        V ob2;
        public Gen(T t, V v)
        {
            ob1 = t;
            ob2 = v;
        }
    }
    class MultipleConstraintDemo
    {
        static void Main()
        {
            // This is OK because string is a class and
            // int is a value type.
            Gen<string, int> obj = new Gen<string, int>("test", 11);
            // The next line is wrong because bool is not
            // a reference type.
            // Gen<bool, int> obj = new Gen<bool, int>(true, 11);
        }
    }
}
```

Creating a Default Value of a Type Parameter

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace InterfaceConstraint
{
    // Demonstrate the default operator.
    class MyClass
    {
        //...
    }
    // Construct a default value of T.
    class Test<T>
    {
        public T obj;
        public Test()
        {
            // The following statement would work only for reference types.
            // obj = null; // can’t use
            // The following statement will work only for numeric value types.
            // obj = 0; // can’t use
            // This statement works for both reference and value types.
            obj = default(T); // Works!
        }
        // ...
    }
    class DefaultDemo
    {
        static void Main()
        {
            // Construct Test using a reference type.
            Test<MyClass> x = new Test<MyClass>();
            if (x.obj == null)
                Console.WriteLine("x.obj is null.");
            // Construct Test using a value type.
            Test<int> y = new Test<int>();
            if (y.obj == 0)
                Console.WriteLine("y.obj is 0.");
        }
    }
}
```

Kết quả:

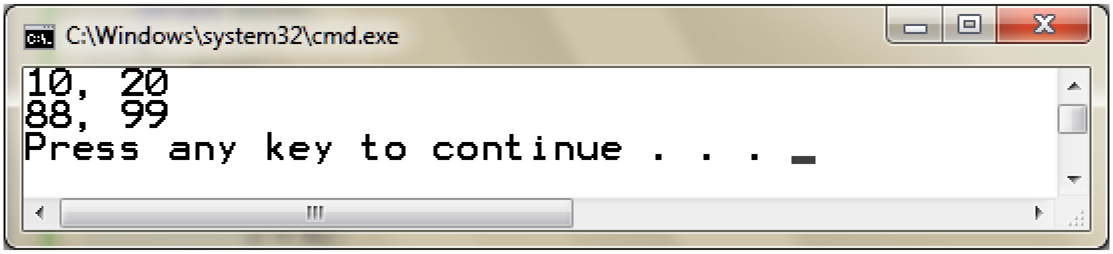


```
x.obj is null.
y.obj is 0.
Press any key to continue . . . _
```

Generic Structures

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace GenericStructures
{
    // Demonstrate a generic struct.
    using System;
    // This structure is generic.
    struct XY<T>
    {
        T x;
        T y;
        public XY(T a, T b)
        {
            x = a;
            y = b;
        }
        public T X
        {
            get { return x; }
            set { x = value; }
        }
        public T Y
        {
            get { return y; }
            set { y = value; }
        }
    }
    class StructTest
    {
        static void Main()
        {
            XY<int> xy = new XY<int>(10, 20);
            XY<double> xy2 = new XY<double>(88.0, 99.0);
            Console.WriteLine(xy.X + ", " + xy.Y);
            Console.WriteLine(xy2.X + ", " + xy2.Y);
        }
    }
}
```

Kết quả:



```
10, 20
88, 99
Press any key to continue . . . _
```

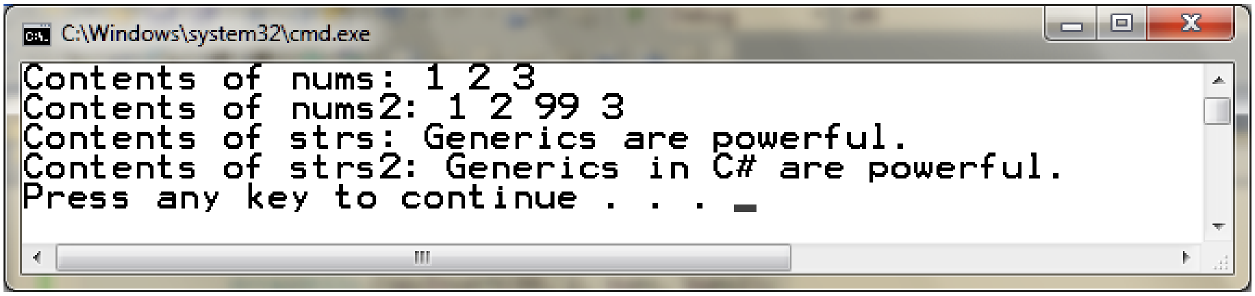
Creating a Generic Method

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace GenericMethods
{
```

```
// Demonstrate a generic method.
// A class of array utilities. Notice that this is not
// a generic class.
class ArrayUtils
{
    // Copy an array, inserting a new element
    // in the process. This is a generic method.
    public static bool CopyInsert<T>(T e, uint idx,
    T[] src, T[] target)
    {
        // See if target array is big enough.
        if (target.Length < src.Length + 1)
            return false;
        // Copy src to target, inserting e at idx in the process.
        for (int i = 0, j = 0; i < src.Length; i++, j++)
        {
            if (i == idx)
            {
                target[j] = e;
                j++;
            }
            target[j] = src[i];
        }
        return true;
    }
}
class GenMethDemo
{
    static void Main()
    {
        int[] nums = { 1, 2, 3 };
        int[] nums2 = new int[4];
        // Display contents of nums.
        Console.Write("Contents of nums: ");
        foreach (int x in nums)
            Console.Write(x + " ");
        Console.WriteLine();
        // Operate on an int array.
        ArrayUtils.CopyInsert(99, 2, nums, nums2);
        // Display contents of nums2.
        Console.Write("Contents of nums2: ");
        foreach (int x in nums2)
            Console.Write(x + " ");
        Console.WriteLine();
        // Now, use copyInsert on an array of strings.
        string[] strs = { "Generics", "are", "powerful." };
```

```
            string[] strs2 = new string[4];
            // Display contents of strs.
            Console.Write("Contents of strs: ");
            foreach (string s in strs)
                Console.Write(s + " ");
            Console.WriteLine();
            // Insert into a string array.
            ArrayUtils.CopyInsert("in C#", 1, strs, strs2);
            // Display contents of strs2.
            Console.Write("Contents of strs2: ");
            foreach (string s in strs2)
                Console.Write(s + " ");
            Console.WriteLine();
            // This call is invalid because the first argument
            // is of type double, and the third and fourth arguments
            // have element types of int.
            // ArrayUtils.CopyInsert(0.01, 2, nums, nums2);
        }
    }
}
```

Kết quả:

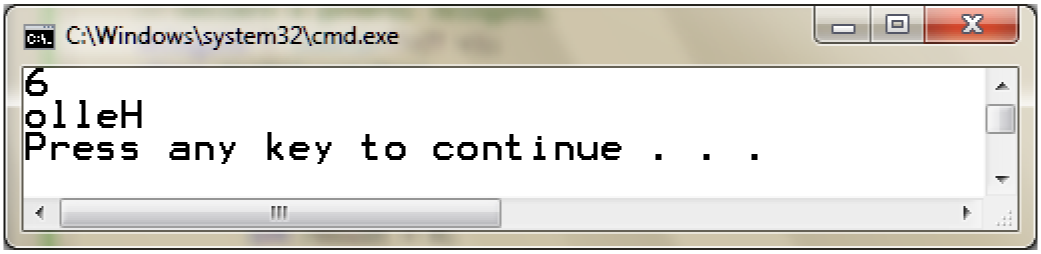


```
Contents of nums: 1 2 3
Contents of nums2: 1 2 99 3
Contents of strs: Generics are powerful.
Contents of strs2: Generics in C# are powerful.
Press any key to continue . . .
```

Generic Delegates

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace GenericDelegates
{
    // A simple generic delegate.
    // Declare a generic delegate.
    delegate T SomeOp<T>(T v);
    class GenDelegateDemo
    {
        // Return the summation of the argument.
        static int Sum(int v)
        {
            int result = 0;
            for (int i = v; i > 0; i--)
                result += i;
            return result;
        }
        // Return a string containing the reverse of the argument.
        static string Reflect(string str)
        {
            string result = "";
            foreach (char ch in str)
                result = ch + result;
            return result;
        }
        static void Main()
        {
            // Construct an int delegate.
            SomeOp<int> intDel = Sum;
            Console.WriteLine(intDel(3));
            // Construct a string delegate.
            SomeOp<string> strDel = Reflect;
            Console.WriteLine(strDel("Hello"));
        }
    }
}
```

Kết quả:



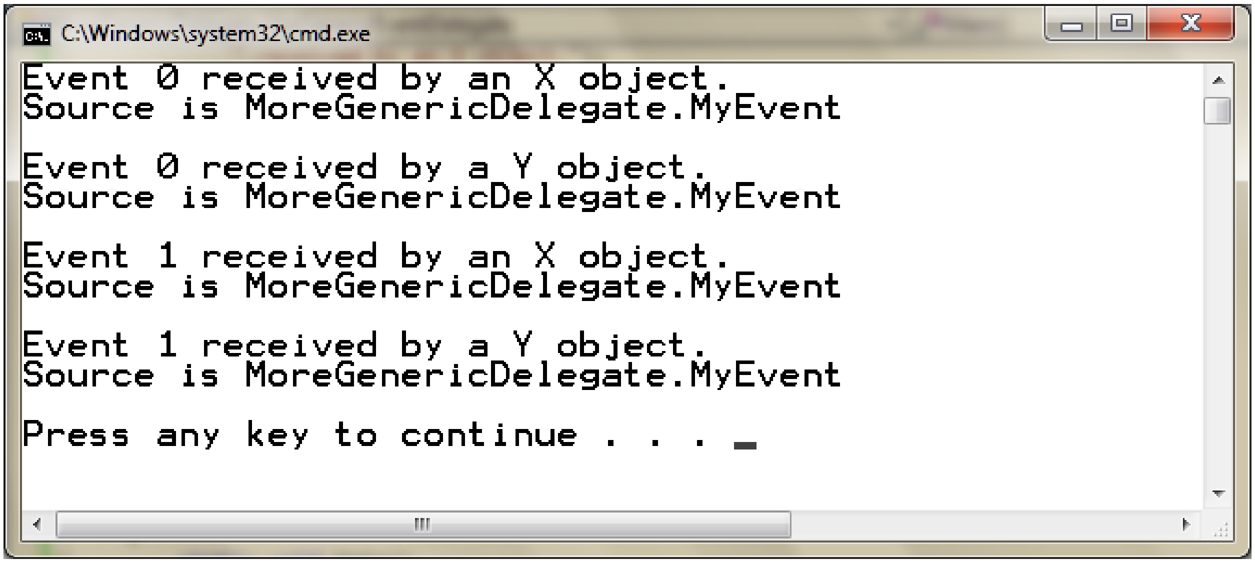
```
6
olleH
Press any key to continue . . .
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace MoreGenericDelegate
{
    // Convert event example from Chapter 15 to use generic delegate.
    using System;
```

```
// Derive a class from EventArgs.
class MyEventArgs : EventArgs
{
    public int EventNum;
}
// Declare a generic delegate for an event.
delegate void MyEventHandler<T, V>(T source, V args);
// Declare an event class.
class MyEvent
{
    static int count = 0;
    public event MyEventHandler<MyEvent, MyEventArgs> SomeEvent;
    // This fires SomeEvent.
    public void OnSomeEvent()
    {
        MyEventArgs arg = new MyEventArgs();
        if (SomeEvent != null)
        {
            arg.EventNum = count++;
            SomeEvent(this, arg);
        }
    }
}
class X
{
    public void Handler<T, V>(T source, V arg) where V : MyEventArgs
    {
        Console.WriteLine("Event " + arg.EventNum +
        " received by an X object.");
        Console.WriteLine("Source is " + source);
        Console.WriteLine();
    }
}
class Y
{
    public void Handler<T, V>(T source, V arg) where V : MyEventArgs
    {
        Console.WriteLine("Event " + arg.EventNum +
        " received by a Y object.");
        Console.WriteLine("Source is " + source);
        Console.WriteLine();
    }
}
class UseGenericEventDelegate
{
    static void Main()
```

```
        {
            X ob1 = new X();
            Y ob2 = new Y();
            MyEvent evt = new MyEvent();
            // Add Handler() to the event list.
            evt.SomeEvent += ob1.Handler;
            evt.SomeEvent += ob2.Handler;
            // Fire the event.
            evt.OnSomeEvent();
            evt.OnSomeEvent();
        }
    }
}
```

Kết quả:



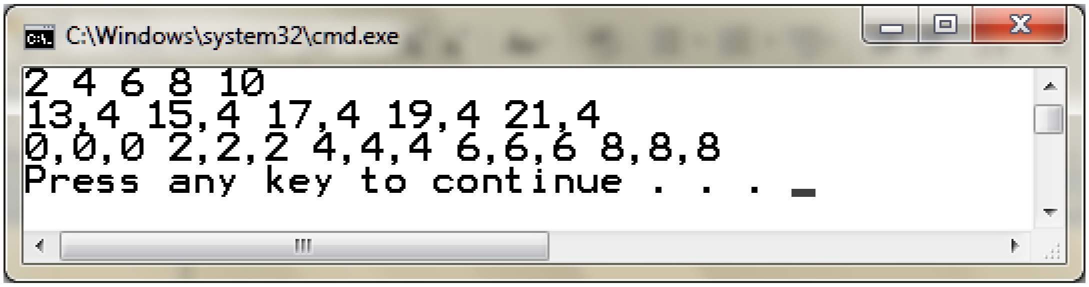
```
Event 0 received by an X object.
Source is MoreGenericDelegate.MyEvent

Event 0 received by a Y object.
Source is MoreGenericDelegate.MyEvent

Event 1 received by an X object.
Source is MoreGenericDelegate.MyEvent

Event 1 received by a Y object.
Source is MoreGenericDelegate.MyEvent

Press any key to continue . . . _
```

Generic Interfaces

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace GenericInterfaces
{
    // Demonstrate a generic interface.
    using System;
    public interface ISeries<T>
    {
        T GetNext(); // return next element in series
        void Reset(); // restart the series
        void SetStart(T v); // set the starting element
    }
```

```
// Implement ISeries.
class ByTwos<T> : ISeries<T>
{
    T start;
    T val;
    // This delegate defines the form of a method
    // that will be called when the next element in
    // the series is needed.
    public delegate T IncByTwo(T v);
    // This delegate reference will be assigned the
    // method passed to the ByTwos constructor.
    IncByTwo incr;
    public ByTwos(IncByTwo incrMeth)
    {
        start = default(T);
        val = default(T);
        incr = incrMeth;
    }
    public T GetNext()
    {
        val = incr(val);
        return val;
    }
    public void Reset()
    {
        val = start;
    }
    public void SetStart(T v)
    {
        start = v;
        val = start;
    }
}
class ThreeD
{
    public int x, y, z;
    public ThreeD(int a, int b, int c)
    {
        x = a;
        y = b;
        z = c;
    }
}
class GenIntfDemo
{
    // Define plus two for int.
    static int IntPlusTwo(int v)
    {
        return v + 2;
    }
    // Define plus two for double.
    static double DoublePlusTwo(double v)
    {
        return v + 2.0;
    }
    // Define plus two for ThreeD.
    static ThreeD ThreeDPlusTwo(ThreeD v)
    {
            if (v == null) return new ThreeD(0, 0, 0);
            else return new ThreeD(v.x + 2, v.y + 2, v.z + 2);
        }
        static void Main()
        {
            // Demonstrate int series.
            ByTwos<int> intBT = new ByTwos<int>(IntPlusTwo);
            for (int i = 0; i < 5; i++)
                Console.Write(intBT.GetNext() + " ");
            Console.WriteLine();
            // Demonstrate double series.
            ByTwos<double> dblBT = new ByTwos<double>(DoublePlusTwo);
            dblBT.SetStart(11.4);
            for (int i = 0; i < 5; i++)
                Console.Write(dblBT.GetNext() + " ");
            Console.WriteLine();
            // Demonstrate ThreeD series.
            ByTwos<ThreeD> ThrDBT = new ByTwos<ThreeD>(ThreeDPlusTwo);
            ThreeD coord;
            for (int i = 0; i < 5; i++)
            {
                coord = ThrDBT.GetNext();
                Console.Write(coord.x + "," +
                coord.y + "," +
                coord.z + " ");
            }
            Console.WriteLine();
        }
    }
}
```

Kết quả:

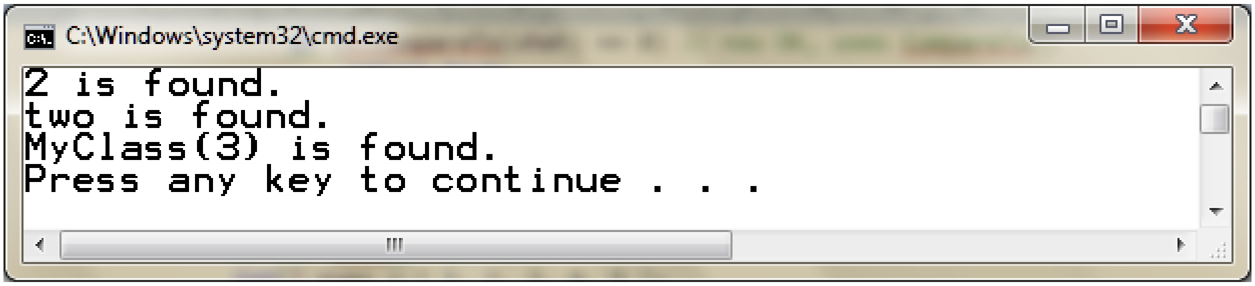


```
2 4 6 8 10
13,4 15,4 17,4 19,4 21,4
0,0,0 2,2,2 4,4,4 6,6,6 8,8,8
Press any key to continue . . .
```

Comparing Instances of a Type Parameter

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace CompareInstanceTypeParameter
{
    // Demonstrate IComparable.

    class MyClass : IComparable
    {
        public int Val;
        public MyClass(int x) { Val = x; }
        // Implement IComparable.
        public int CompareTo(object obj)
        {
            return Val - ((MyClass)obj).Val;
        }
    }
    class CompareDemo
    {
        // Require IComparable interface.
        public static bool IsIn<T>(T what, T[] obs) where T : IComparable
        {
            foreach (T v in obs)
                if (v.CompareTo(what) == 0) // now OK, uses CompareTo()
                    return true;
            return false;
        }
        // Demonstrate comparisons.
        static void Main()
        {
            // Use IsIn() with int.
            int[] nums = { 1, 2, 3, 4, 5 };
            if (IsIn(2, nums))
                Console.WriteLine("2 is found.");
            if (IsIn(99, nums))
                Console.WriteLine("This won’t display.");
            // Use IsIn() with string.
            string[] strs = { "one", "two", "Three" };
            if (IsIn("two", strs))
                Console.WriteLine("two is found.");
            if (IsIn("five", strs))
                Console.WriteLine("This won’t display.");
            // Use IsIn with MyClass.
            MyClass[] mcs = { new MyClass(1), new MyClass(2), new MyClass(3), new
MyClass(4) };
            if (IsIn(new MyClass(3), mcs))
                Console.WriteLine("MyClass(3) is found.");
            if (IsIn(new MyClass(99), mcs))
                Console.WriteLine("This won’t display.");
        }
    }
}
```

Kết quả:

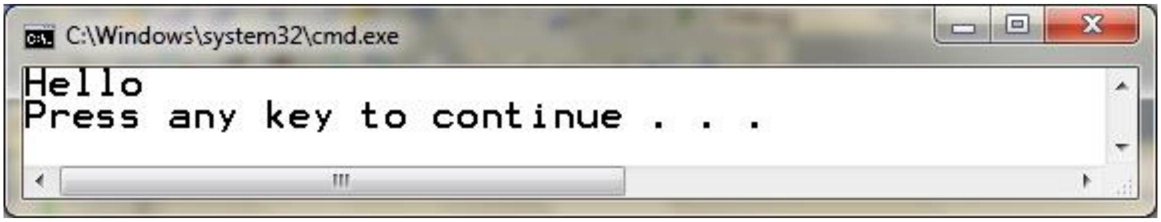


```
2 is found.
two is found.
MyClass(3) is found.
Press any key to continue . . .
```

Generic Class Hierarchies
Using a Generic Base Class

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace UsingaGenericBaseClass
{
    // A simple generic class hierarchy.
    // A generic base class.
    class Gen<T>
    {
        T ob;
        public Gen(T o)
        {
            ob = o;
        }
        // Return ob.
        public T GetOb()
        {
            return ob;
        }
    }
    // A class derived from Gen.
    class Gen2<T> : Gen<T>
    {
        public Gen2(T o)
            : base(o)
        {
            // ...
        }
    }
    class GenHierDemo
    {
        static void Main()
        {
            Gen2<string> g2 = new Gen2<string>("Hello");
            Console.WriteLine(g2.GetOb());
        }
    }
}
```

Kết quả:

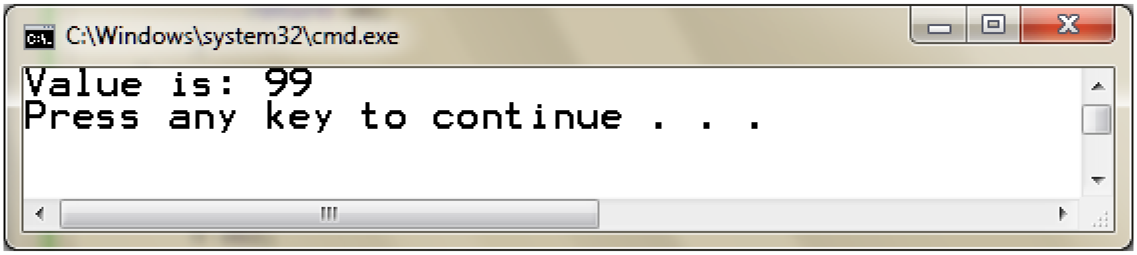


```
Hello
Press any key to continue . . .
```

Thí dụ khác:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace UsingaGenericBaseClass2
{
    // A derived class can add its own type parameters.
    // A generic base class.
    class Gen<T>
    {
        T ob; // declare a variable of type T
        // Pass the constructor a reference of type T.
        public Gen(T o)
        {
            ob = o;
        }
        // Return ob.
        public T GetOb()
        {
            return ob;
        }
    }
    // A derived class of Gen that defines a second
    // type parameter, called V.
    class Gen2<T, V> : Gen<T>
    {
        V ob2;
        public Gen2(T o, V o2)
            : base(o)
        {
            ob2 = o2;
        }
        public V GetObj2()
        {
            return ob2;
        }
    }
    // Create an object of type Gen2.
    class GenHierDemo2
    {
        static void Main()
        {
            // Create a Gen2 object for string and int.
            Gen2<string, int> x =
            new Gen2<string, int>("Value is: ", 99);
            Console.Write(x.GetOb());
            Console.WriteLine(x.GetObj2());
        }
    }
}
```

Kết quả:

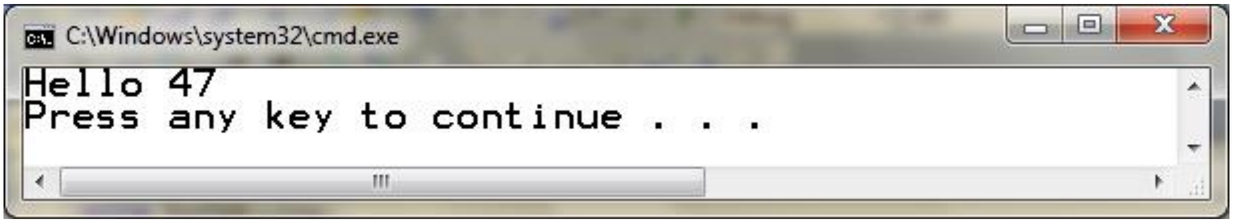


```
Value is: 99
Press any key to continue . . .
```

A Generic Derived Class

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace AGenericDerivedClass
{
    // A non-generic class can be the base class of a generic derived class.
    // A non-generic class.
    class NonGen
    {
        int num;
        public NonGen(int i)
        {
            num = i;
        }
        public int GetNum()
        {
            return num;
        }
    }
    // A generic derived class.
    class Gen<T> : NonGen
    {
        T ob;
        public Gen(T o, int i)
            : base(i)
        {
            ob = o;
        }
        // Return ob.
        public T GetOb()
        {
            return ob;
        }
    }
    // Create a Gen object.
    class HierDemo3
    {
        static void Main()
        {
            // Create a Gen object for string.
            Gen<String> w = new Gen<String>("Hello", 47);
            Console.Write(w.GetOb() + " ");
            Console.WriteLine(w.GetNum());
        }
    }
}
```

Kết quả:

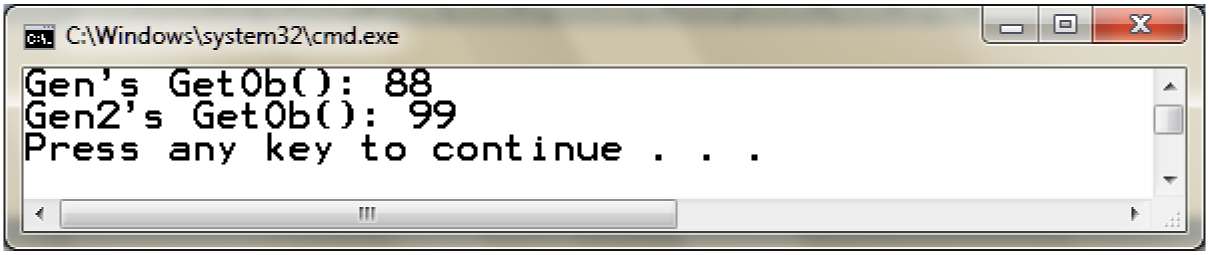


```
Hello 47
Press any key to continue . . .
```

Overriding Virtual Methods in a Generic Class

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace OverridingVirtualMethods
{
    // Overriding a virtual method in a generic class.
    // A generic base class.
    class Gen<T>
    {
        protected T ob;
        public Gen(T o)
        {
            ob = o;
        }
        // Return ob. This method is virtual.
        public virtual T GetOb()
        {
            Console.Write("Gen’s GetOb(): ");
            return ob;
        }
    }
    // A derived class of Gen that overrides GetOb().
    class Gen2<T> : Gen<T>
    {
        public Gen2(T o) : base(o) { }
        // Override GetOb().
        public override T GetOb()
        {
            Console.Write("Gen2’s GetOb(): ");
            return ob;
        }
    }
    // Demonstrate generic method override.
    class OverrideDemo
    {
        static void Main()
        {
            // Create a Gen object for int.
            Gen<int> iOb = new Gen<int>(88);
            // This calls Gen’s version of GetOb().
            Console.WriteLine(iOb.GetOb());
            // Now, create a Gen2 object and assign its
            // reference to iOb (which is a Gen<int> variable).
            iOb = new Gen2<int>(99);
            // This calls Gen2’s version of GetOb().
            Console.WriteLine(iOb.GetOb());
        }
    }
}
```

Kết quả:



```
Gen's GetOb(): 88
Gen2's GetOb(): 99
Press any key to continue . . .
```

Overloading Methods That Use Type Parameters

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace OverloadingMethodsThatUseTypeParameters
{
    // Ambiguity can result when overloading methods that
    // use type parameters.
    //
    // This program will not compile.
    using System;
    // A generic class that contains a potentially ambiguous
    // overload of the Set() method.
    class Gen<T, V>
    {
        T ob1;
        V ob2;
        // ...
        // In some cases, these two methods
        // will not differ in their parameter types.
        public void Set(T o)
        {
            ob1 = o;
        }
        public void Set(V o)
        {
            ob2 = o;
        }
    }
    class AmbiguityDemo
    {
        static void Main()
        {
            Gen<int, double> ok = new Gen<int, double>();
            Gen<int, int> notOK = new Gen<int, int>();
            ok.Set(10); // is valid, type args differ
            notOK.Set(10); // ambiguous, type args are the same!
        }
    }
}
```