**C#\_9.1 Generics**

At its core, the term generics means parameterized types. *Parameterized types* are important because they enable you to create classes, structures, interfaces, methods, and delegates in which the type of data upon which they operate is specified as a parameter. In pre-generics code, casts were needed to convert between the object type and the actual type of the data. *Generics* add the type safety that was lacking because it is no longer necessary to employ a cast to translate between object and the actual data type.

* Generics class: The following program defines two classes. The first is the generic class MyGenClass, and the second is GenericsDemo, which uses MyGenClass.

|  |  |
| --- | --- |
| * Here, T is the name of a type parameter. This name is used as a placeholder for the actual type that will be specified when a MyGenClass object is created. Whenever a type parameter is being declared, it is specified within angle brackets. ob will be a variable of the type bound to T when a MyGenClass object is instantiated. For example, if type string is specified for T, then in that instance, ob will be of type string. * ***MyGenClass<int> iOb;*** creates a version of MyGenClass for type int. * The type int is specified within the angle brackets after MyGenClass. In this case, int is a type argument that is bound to MyGenClass’s type parameter, T. This creates a version of MyGenClass in which all uses of T are replaced by int. Thus, for this declaration, ob is of type int, and the return type of **GetOb()** is of type int. | **using System;**  */\* Here, MyGenClass is a generic class that has one type parameter called T.*  *T will be replaced by a real type when a MyGenClass object is constructed. \*/*  **class** MyGenClass<T> { **T** ob; *// declare a variable of type T*  **public** MyGenClass(T o) { ob = o; } *// constructor has a parameter of type T.*  **public** **T** GetOb() { **return** ob; } *// Return ob, which is of type T.*  }  **class** GenericsDemo { **static void Main()** {  **MyGenClass<int>** iOb; *// Declare a MyGenClass reference for int.*  iOb = **new MyGenClass<int>**(88); *// Create a MyGenClass<int> object.*  **int** v = iOb.GetOb(); *// Get the value in iOb*.  **Console.WriteLine**( v + "\n");  *// Create a MyGenClass object for strings*.  **MyGenClass<string>** strOb = **new** MyGenClass<string>("Generics ");  **string** str = strOb.GetOb(); *// Get the value in strOb.*  **Console.WriteLine**(str + "\n"); }} |

* Closed constructed type: When you specify a *type argument* such as *int* or string for *MyGenClass*, you are creating what is referred to in C# as a *closed* *constructed type*. Thus, *MyGenClass<int>* is a *closed constructed type*.
* open constructed type: In essence, a generic type, such as MyGenClass<T>, is an abstraction. It is only after a *specific version*, such as MyGenClass<int>, has been constructed that a concrete type has been created. In C# terminology, a construct such as MyGenClass<T> is called an open constructed type, because T *(rather than an actual type such as int)* is specified.
* ***iOb = new MyGenClass<int>(88);*** assigns to iOb a reference to an instance of an int version of the MyGenClass class. Notice that when the MyGenClass constructor is called, the type argument int is also specified. This is necessary because the type of the variable (in this case, iOb) to which the reference is being assigned is of type MyGenClass<int>. Thus, the reference returned by new must also be of type MyGenClass<int>. If it isn’t, a compile-time error will result. For example,

***iOb = new MyGenClass<byte>(16); // Error! Wrong Type!***

* *A reference of one specific version of a generic type is* not type-compatible *with another version of the same generic type*. For example, ***iOb = strOb;*** is in error and will not compile. Even though both iOb and strOb are of type MyGenClass<T>, they are references to different types because their type arguments differ.
* A Generic Class with Two Type Parameters: For Example TwoGen can be declared: **class TwoGen<T, V> {**
* It specifies two *type parameters*, *T* and *V*, separated by a *comma*. Because it has two type parameters, *two type arguments* must be specified for *TwoGen* when an *object* is *created*, as: ***TwoGen<int, string> tgObj = new TwoGen<int, string>(1024, "Using two type parameters");***
* In this case, int is substituted for T and string is substituted for V.
* Although the two type arguments differ in this example, it is possible for both types to be the same. For example, the following line of code is valid:

***TwoGen<double, double> x = new TwoGen<double, double>(98.6, 102.4);***

* In this case, both *T* and *V* are of type *double*. Of course, if *the type arguments* were *always the same*, then *two type parameters* would be *unnecessary*.
* GENERALIZED SYNTAX: 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> { // ...**

* Here is the syntax for declaring a *reference* to a *generic class* and giving it an *initial* value:

**class-name<type-arg-list> var-name = new class-name<type-arg-list>(cons-arg-list);**

**C#\_9.2 Generics Improve Type Safety *(Type safe code)***

Generics automatically ensure the type safety of all operations involving MyGenClass. Generics eliminate the need for you to use casts and to type-check code by hand. To understand the benefits of generics, first consider the following program that creates a non-generic equivalent of MyGenClass called NotGeneric:

|  |  |
| --- | --- |
| * Notice that NotGeneric replaces all uses of T with object. This makes NotGeneric able to store any type of object, as can the generic version. However, this ***is bad for two reasons***. * First, explicit casts must be employed to retrieve the stored data. Notice this line:   ***int v = (int) iOb.GetOb();***   * Because the return type of ***GetOb()*** is now object, the cast to int is necessary to enable the value returned by ***GetOb()*** to be unboxed and stored in v. If you remove the cast, the program will not compile. * In the generic version of the program, this cast was not needed because ***int*** was specified as a type argument when ***iOb*** was constructed. * Second, many kinds of type mismatch errors *cannot be found until runtime*. Consider the following sequence from near the end of the program.   **iOb = strOb;** *// This compiles, but is conceptually wrong!*  *//* **v = (int) iOb.GetOb();** *// results in a runtime exception.* | *// NotGeneric is functionally equivalent to MyGenClass but does not use generics.*  using System;  class NotGeneric { object ob;  public NotGeneric(object o) { ob = o; }  public object GetOb() { return ob; } }  class NonGenDemo { static void Main() {  NotGeneric iOb = new NotGeneric(88); *// NotGeneric object.*  int v = (int) iOb.GetOb(); *// Get the value in iOb, a cast is necessary.*  Console.WriteLine( v + "\n");  NotGeneric strOb = new NotGeneric("Non-Generic "); *// NotGeneric obj*  String str = (string) strOb.GetOb(); *// cast is necessary.*  Console.WriteLine( str + "\n");  iOb = strOb; *// This compiles, but is conceptually wrong!*  *// The following line results in a runtime exception.*  *//* v = (int) iOb.GetOb(); */\* runtime error: Runtime type mismatch \*/* }} |

* Here, strOb is assigned to iOb. However, strOb refers to an object that contains a string, not an integer. This assignment is *syntactically valid* because all NotGeneric references are of the same type. Thus, any NotGeneric reference can refer to any NotGeneric object.
* However, the statement is *semantically wrong*. In that line, the *return type* of **GetOb()** is *cast* to **int** and then an attempt is made to *assign* this value to **v**.
* The trouble is that *iOb* now refers to an *object* that stores a *string*, not an *int*. Unfortunately, *without the use* of *generics*, the *compiler won’t catch* this *error*. Instead, a *runtime exception* will *occur* when the *cast* to *int* is attempted.
* The preceding sequence can’t occur when generics are used. If this sequence were attempted in the generic version of the program, the compiler would catch it and report an error, thus preventing a serious bug that results in a runtime exception.
* Note: Properties, operators, indexers, and events cannot declare type parameters. Thus, they *cannot be made generic*. However,

they can be used in a *generic class* and make use of the *type parameters* defined by that *class*.

**C#\_9.3 Constrained Types (Similar to Java's Bounded types)**

C# provides constrained types, which is similar to Java's Bounded types. When specifying a type parameter, you can specify a constraint that the type parameter must satisfy. This is accomplished through the use of a ***where*** clause when specifying the type parameter, as shown here:

***class class-name<type-param> where type-param : constraints { /\* . . . \*/*** Here, *constraints* is a *comma-separated list of constraints*.

|  |  |
| --- | --- |
| C# defines the following types of constraints | |
| BASE CLASS constraint | You can *require* that a *certain base class* be present in a *type argument* by using a *base class constraint*. This constraint is ***specified*** ***by*** naming the *desired base class*.   * There is a variation of this constraint, called a ***naked type constraint***, in which the *base class* is specified as a *type parameter* rather than *an* *actual type*. This enables you to establish a relationship between two type parameters. |
| INTERFACE constraint | You can require that ***one*** or ***more*** *interfaces* be implemented by a *type argument* by using an *interface constraint*. This constraint is ***specified*** ***by*** naming the *desired interface*. |
| CONSTRUCTOR constraint | You can *require* that the *type argument* supply a *parameterless constructor*. It is called a *constructor constraint*. It is ***specified by*** ***new()***. |
| REFERENCE type constraint | You can *specify* that a *type argument* must be a *reference type* ***by specifying*** the reference type constraint: ***class***. |
| VALUE type constraint | You can *specify* that the *type argument* be a *value type* ***by specifying*** the value type constraint: ***struct***. |
| * Of these constraints, the base class constraint and the interface constraint are probably the most often used, but all are important. | |

* Base Class Constraint: The base class constraint enables you to specify a base class that a type argument must inherit. Itserves two important purposes:
* A base class constraint enables a generic class to access the members of the base class. It also ensures that *only those type arguments* that fulfill this constraint are valid, thus preserving *type-safety*.
* First, it lets you use the members of the base class specified by the constraint within the generic class. For example, you can call a *method* or use a *property* of the *base*. By supplying a *base class constraint*, you are letting the *compiler* know that *all type arguments* will have the *members* defined by the *base class constraint*.
* The second purpose of a base class constraint is to ensure that only *type arguments* that *support the specified base class can be used*. This means that for any given base class constraint, the type argument must be either the *base itself* or a *class* derived from that *base*. If you attempt to use a *type argument* that does not *match* or *inherit* the specified *base* class, a *compile-time error* will result.
* The base class constraint uses this form of the ***where*** clause: ***where T : base-class-name***
* Here, *T* is the *name* of the ***type parameter***, and *base-class-name* is the name of the ***base class***. *Only one base class can be specified*.
* C#\_Example 1: Following demonstrates the base class constraint mechanism. It creates a *base* called *MyStrMethods*, which defines a *public* *method* called ***ReverseStr()*** that returns a reversed version of its string argument. So, any *derived* class of *MyStrMethods* will have access to this method.

|  |  |
| --- | --- |
| using System;  class MyStrMethods {  public string ReverseStr(string str) { string result = "";  foreach(char ch in str) result = ch + result;  return result; }  */\* . . . \*/* }  class MyClass : MyStrMethods { } *// MyClass inherits MyStrMethods.*  class MyClass2 { } *// MyClass2 does not inherit MyStrMethods.*  */\* Because of the base class constraint, all type arguments specified for Test must have MyStrMethods as a base class. \*/*  class Test<T> where T : MyStrMethods {  T obj;  public Test(T o) { obj = o; }  public void ShowReverse(string str) { string revStr = obj.ReverseStr(str);  Console.WriteLine(revStr); }  *// OK to call* ***ReverseStr()*** *on obj because it's declared by the base class* ***MyStrMethods****.*  } | class BaseClassConstraintDemo { static void Main() {  MyStrMethods objA = new MyStrMethods();  MyClass objB = new MyClass();  MyClass2 objC = new MyClass2();  *// The following is valid because MyStrMethods is the specified base class.*  Test<MyStrMethods> t1 = new Test<MyStrMethods>(*objA*);  t1.ShowReverse("This is a test.");  *// The following is valid because MyClass inherits MyStrMethods.*  Test<MyClass> t2 = new Test<MyClass>(*objB*);  t2.ShowReverse("More testing.");  *// The following is invalid because MyClass2 DOES NOT*  *// inherit MyStrMethods.*  *//* Test<MyClass2> t3 = new Test<MyClass2>(*objC*); *// Error!*  *//* t3.ShowReverse("Error!");  }} |

* In this program, the class *MyStrMethods* is inherited by *MyClass*, but *not* by *MyClass2*.
* Next, notice that **Test** is a generic class that is declared like this: ***class Test<T> where T : MyStrMethods {***
* The where clause ensures that any type argument specified for T must have MyStrMethods as a base class.
* As you can see, the *object* passed to ***Test()*** is stored in *obj*. Now notice that ***Test*** declares the method ***ShowReverse()***, shown next:

**public void ShowReverse(string str){ string revStr = obj.ReverseStr(str); Console.WriteLine(revStr); }**

* This method calls ***ReverseStr()*** on *obj*, which is a *T* object, and then displays the reversed string. The key point is that the only reason that ***ReverseStr()*** can be called is because the *base* class constraint requires that any type argument bound to *T* will inherit *MyStrMethods*, which declares ***ReverseStr()***. If the *base class constraint* had not been used, the compiler wouldn’t know that a method called ***ReverseStr()*** can be called on an *object* of type *T*.
* In addition to *enabling access* to *members* of the *base* class, the *base* *class constraint* enforces that only *types* that *inherit* the *base* class can be used as *type* *arguments*. This is why the following two lines are commented-out:

**// Test<MyClass2> t3 = new Test<MyClass2>(objC); // Error!**

**// t3.ShowReverse("Error!");**

* Because *MyClass2* does not inherit *MyStrMethods*, it can’t be used as a *type argument* when constructing a *Test* object.

|  |  |
| --- | --- |
| * Generic method with multiple constraints (stack overflow): A generic class Test which has two type parameters T, V then its constraint for two different base would be:   **public Test< T, V > where T : Tbase where V : Vbase{}**   * multiple parameters, multiple constraints to a single parameter: You can apply constraints to multiple parameters, and multiple constraints to a single parameter, as shown in the example on right side: | **class Base { }**  **class Test<T, U>**  **where U : struct**  **where T : Base, new()**  **{ }** */\** ***Microsoft*** *\*/* |

* NAKED type constraint(Use a Constraint to Establish a Relationship Between Two Type Parameters): There is a variation of the base class constraint that allows you to establish a relationship between two ***type parameters***. Eg: consider: ***class MyGenClass<T, V> where V : T {}***
* This *constraint* requires that the *type argument* passed to *T* must be a *base* class of the *type argument* passed to *V*. In this declaration, the *where* clause tells the *compiler* that the *type argument* bound to *V* must be *identical* to or *inherit* from the *type argument* bound to *T*. If this relationship is not present, a ***compile-time error*** will result. A constraint that uses a type parameter such as that just shown is called a ***naked type constraint***. For example:

|  |  |
| --- | --- |
| class A { */\* . . . \*/*  }  class B : A { */\* . . . \*/*  }  class MyGenClass<T, V> where V : T { */\* . . . \*/* } *// Here, V must inherit T.* | class NakedConstraintDemo { static void Main() {  MyGenClass<A, B> x = new MyGenClass<A, B>(); *// This declaration is OK because B inherits A.*  *//* MyGenClass<B, A> y = new MyGenClass<B, A>(); */\* Results error since, A does not inherit B.\*/* }} |

* Notice that class B inherits class A, hence **MyGenClass<A, B> x = new MyGenClass<A, B>();** is legal.
* However, the second declaration: **// MyGenClass<B, A> y = new MyGenClass<B, A>();** is illegal because A does not inherit B.
* Interface Constraint: The interface constraint enables you to specify an interface that a ***type argument must implement***. The interface constraint serves the same two purposes as the base class constraint. The interface constraint uses this form of the where clause: **where T : interface-name**
* T is the name of the ***type parameter***, and interface-name is the ***name of the interface***. More than one interface can be specified by using a comma-separated list.
* If a *constraint* includes *both* a *base* class and *interface*, then the *base* class must be listed *first*. For example,

|  |  |
| --- | --- |
| interface IMyInterface { void Start(); void Stop(); }  class MyClass : IMyInterface { public void Start() { Console.WriteLine("Starting..."); }  public void Stop() { Console.WriteLine("Stopping..."); }  }  class MyClass2 { } // Class MyClass2 does not implement IMyInterface.  */\* Because of the interface constraint, all type arguments specified for Test must implement IMyInterface \*/*  class Test<T> where T : IMyInterface { T obj;  public Test(T o) { obj = o; }  public void Activate() { *// OK to call Start() and Stop(); since they're declared by IMyInterface.*  obj.Start(); obj.Stop(); } } | class InterfaceConstraintDemo { static void Main() {  MyClass objA = new MyClass();  MyClass2 objB = new MyClass2();  */\* The following is valid because MyClass implements IMyInterface. \*/*  Test<MyClass> t1 = new Test<MyClass>(objA);  t1.Activate();  */\* The following is invalid because MyClass2 DOES NOT implement IMyInterface. \*/*  *//*  Test<MyClass2> t2 = new Test<MyClass2>(objB);  *//*  t2.Activate();  }} |

* Notice, MyClass, implements IMyInterface. The second class, MyClass2, does not.
* The ***generic class*** Test uses an interface constraint to require that T implement the interface IMyInterface. Also notice that an object of type T is passed to Test’s constructor and stored in obj. Test defines a method called ***Activate()***, which uses ***obj*** to call the ***Start()*** and ***Stop()*** methods declared by IMyInterface.
* Because of the interface constraint, all type arguments specified for Test must implement IMyInterface.
* ***Test<MyClass> t1 = new Test<MyClass>(objA);*** is valid because *MyClass* implements *IMyInterface*, satisfies the *interface constraint*.
* ***Test<MyClass2> t2 = new Test<MyClass2>(objB);*** is invalid because *MyClass2* DOES NOT implement *IMyInterface*, doesn’t satisfy the *interface constraint*.
* new( ) Constructor Constraint: The ***new()*** ***constructor constraint*** enables you to instantiate an object of a generic type. Normally, you *cannot create an instance* of a *generic type parameter*. However, the ***new()*** constraint changes this because it requires that a *type argument* supply a *parameterless constructor*. (This *parameterless* *constructor* can be the default constructor provided automatically when no explicit constructors are declared.)
* With the ***new()*** constraint in place, you can invoke the parameterless constructor to create an object of the generic type. For example:

|  |  |
| --- | --- |
| class MyClass { public MyClass() { Console.WriteLine("Creating a MyClass instance."); */\* . . . \*/* } }  class Test<T> where T : new() { T obj;  public Test() { Console.WriteLine("Creating a Test instance.");  *// following works because of the new() constraint.*  obj = new T(); */\* create a T object \*/* } } | class ConsConstraintDemo {  static void Main() {  Test<MyClass> t = new Test<MyClass>(); }} |
| output: Creating a Test instance.  Creating a MyClass instance. |

* First, notice the declaration of the Test class, shown here: class Test<T> where T : new() {
* Because of the new( ) constraint, any type argument must supply a parameterless constructor, which can be the default constructor or one that you create.

|  |  |
| --- | --- |
| * Next, examine the **Test** constructor, shown here:   **public** Test(){ **Console.WriteLine**("*Creating a Test instance*.");  obj = **new** T(); */\* create a T object \*/* } | * A new object of type ***T*** is created, and a reference to it is assigned to ***obj***. This statement is valid only because the ***new()*** constraint ensures that a constructor will be available. Without the ***new()*** constraint ,an error will be reported. |

* In **Main( )**, an object of type **Test** is instantiated, as: **Test<MyClass> x = new Test<MyClass>();** Notice that the type argument is MyClass and that MyClass defines a parameterless constructor. Thus, it is valid for use as a type argument for Test.
* Remember, *it was not necessary* for MyClass to explicitly declare a parameterless constructor. Its default constructor would also satisfy the constraint.
* But, if a class needs other constructors in addition to a parameterless one, then it would be necessary to also explicitly declare a parameterless version.

|  |  |
| --- | --- |
| * Here are three important points about using ***new()*** : | * *First*, it *can* be used with *other constraints*, but it *must* be the *last constraint* in the *list*. * *Second*, ***new()*** allows you to *construct an object* using only the *parameterless* *constructor*, even when other *constructors* are available. In other words, it is *not permissible to pass* *arguments* to the *constructor* of a *type parameter*. * *Third*, you cannot use ***new()*** in *conjunction* with a *value type constraint*, described next. |

* The Reference Type Value Type Constraints: These two constraints enable you to indicate that a type argument must be either a reference type or a value type. These are useful in the few cases in which the difference between reference and value types is important to generic code.
* General form of the reference type constraint: **where T : class** In this form of the ***where*** clause, the keyword ***class*** specifies that ***T*** must be a ***reference type***.
* Thus, an attempt to use a value type, such as ***int*** or ***bool***, for *T* will result in a *compilation error*.
* General form of the value type constraint: **where T : struct** In this case, the keyword ***struct*** specifies that ***T*** must be a ***value*** ***type***.
* Recall that ***structures*** are ***value types***. Thus, an attempt to use a *reference type*, such as ***string***, for ***T*** result a *compilation error*.
* In both cases, when additional constraints are present, ***class*** or ***struct*** must be the first constraint in the list.

|  |  |  |
| --- | --- | --- |
| Example that demonstrates the reference type constraint. | Example that demonstrate a value type constraint. | |
| **using System**;  **class** MyClass { /\* . . . \*/ }  **class** Test<T> **where** T : class { *// Use a reference constraint.*  **T** obj;  */\* The following statement is legal only because* ***T*** *is guaranteed to be a* ***reference******type****, which can be assigned the value* ***null****. \*/*  **public** Test() { 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**>(); *\*/* }} | **using System**;  **struct** MyStruct { /\* . . . \*/ }  **class** MyClass { /\* . . . \*/ }  **class** Test<T> **where** T : **struct** { *// Use a value-type constraint.*  **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);  *// Following declaration is illegal. CLASS or REFERNCE type .*  */\** **Test<MyClass>** z = **new** Test<*MyClass*>(**new** MyClass()); *\*/* }} | |
| * **class Test<T> where T : class {** The class constraint requires that any type argument for T be a reference type. In this program, this is necessary because of what occurs inside the Test constructor:   **public Test() { obj = null; }**  Here, obj (which is of type T) is assigned the value null. This assignment is valid only for reference types. | * **class Test<T> where T : struct {** Because T of Test now has the struct constraint, T can be bound to only value type arguments. This means that Test<MyStruct> and Test<int> are valid, but Test<MyClass> is results in a compilr-time error. | |
| * As a general rule, you *cannot assign* null to a value type. (The exception to this rule is the nullable type, which is a special structure type that encapsulates a value type and allows the value null. See Chapter 12). i.e, without the constraint, the *assignment* would not have been *valid* and the *compile* would have *failed*. * This is one case in which the difference between *value types* and *reference types* might be *important* to a *generic routine*. | | * The value type constraint is the *complement* of the reference type constraint. It simply ensures that any *type* *argument* is a *value* *type*, including a *struct* or an *enum*. (In this context, a *nullable* *type* is *not considered* a *value* *type*.) |

**C#\_9.4Multiple Constraints: Details (Recall *previous* *section "Stack overflow"* point)**

* One type parameter and multiple constraints: There can be ***more than one constraint*** associated with a (one) parameter. When this is the case, use a comma-separated list of *constraints*.
* In this list, the *first constraint* must be *class* or *struct* (if present), or the *base* class (if one is specified).
* It is illegal to specify both a ***class*** or " ***struct*** constraint and a ***base*** class " constraint.
* Next must be any ***interface*** constraints. The ***new()*** constraint must be last.
* For example, this is a valid declaration: **class** MyGenClass<T> **where** T : **MyClass**, **IMyInterface**, **new()** { // ...
* In this case, T must be replaced by a type argument that *inherits* MyClass, *implements* IMyInterface, and has a parameterless constructor.
* multiple type parameters with different constraints: When using two or more type parameters, you can specify a constraint for each parameter by using a separate where clause.
* For example, following uses *multiple* where *clauses*.

|  |  |
| --- | --- |
| using System;  class TwoWheres<T, V> where T : class  where V : struct { *// Two Wheres has two type arguments*  T ob1; V ob2;  public TwoWheres(T t, V v){ ob1 = t; ob2 = v; } } | class TwoWheresDemo { static void Main() {  *// This is OK because string is a class and int is a value type.*  TwoWheres<string, int> obj = new TwoWheres<string, int>("test", 11);    *// The following* *is* *wrong because bool is not a reference type.*  */\** TwoWheres<bool, int> obj2 = new TwoWheres<bool, int>(true, 11); *\*/* }} |

* In this example, TwoWheres takes two type arguments and both have a where clause. Pay special attention to its declaration:

***class*** *TwoWheres<T, V>* ***where*** *T :* ***class******where*** *V :* ***struct*** *{*

* Notice, the only thing that *separates* the *first* *where* from the second is whitespace "*no comma or semicolon*". *No other punctuation is required or valid*.

**C#\_9.5 DEFAULT VALUE of a Type Parameter**

When writing generic code, there will be times when the difference between value types and parameter types is an issue. One such situation occurs when you want to give a variable of a type parameter a default value.

* For reference types, the default value is null. For non-struct value types, the default value is 0. The default value for a struct is an object of that struct with *all fields set to their defaults*.
* Thus, trouble occurs if you want to give a variable of a type parameter a default value. What value would you use: ***null***, ***0***, or *something else*? For example, given a generic class called Test declared like this: **class Test<T> { T obj; */\* \*/***

To give obj a *default value*, we can use: obj = null; */\* works only for reference types \*/* or obj = 0; */\* works only for numeric types and enums, but not structs \*/*

* default(type): The solution to this problem is to use *another form of default*, shown here: ***default(type)*** This is the operator form of default, and it produces a *default value* of the *specified* type, no matter what type is used. Thus, continuing with the example, to assign obj a default value of type T, you would use this statement: ***obj = default(T);***

This will work for all type arguments, whether they are value or reference types. Here is a short program that demonstrates default:

|  |  |
| --- | --- |
| using System;  class MyClass { /\* . . . \*/ }  *// Construct a default value of T.*  class Test<T> { public T obj;  public Test() {  *//* obj = null; *// can't use // This statement will work only for reference types.*  *//* obj = 0; *// can't use // This statement will work only for numeric value types.*  *//*  *Folloeing statement works for both reference and value types.*  obj = default(T); */\* Create a default value for any T.\*/* }} | 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."); }} |
| output: ***x.obj is null.***  ***y.obj is 0.*** |

|  |  |
| --- | --- |
| Note: | Short generic declaration: Implicitly typed variable feature can *shorten a long declaration* that includes an initializer. Since, in a ***var*** declaration, the *type* of the variable is determined by the type of the *initializer*. Therefore, a declaration such as  **SomeClass<String, bool> someObj = new SomeClass<string, bool>("testing", false);** can be more compactly written as  **var someObj = new SomeClass<string, bool>("testing", false);**  Although the use of ***var*** does shorten the code here, its primary use is with *anonymous types*, which are described in *Chapter 12*. Also, because *implicitly* *typed* *variables* are new to C#, it’s not clear (at the time of this writing) that the preceding use of *var* will be considered a “*best practice*” by all C# practitioners. |

**C#\_9.6 Generic Structures**

You can create a *structure* that takes *type parameters*. The syntax for a *generic structure* is the same as for *generic classes*. For example, in the following program, the *KeyValue* *structure*, which stores key/value pairs, is generic:

|  |  |
| --- | --- |
| using System;  struct KeyValue<TKey, TValue> { *// This structure is generic.*  public TKey key;  public TValue val;  public KeyValue(TKey a, TValue b) { key = a; val = b; }  } | class GenStructDemo { static void Main() {  KeyValue<string, int> kv = new KeyValue<string, int>("Tom", 20);  KeyValue<string, bool> kv2 = new KeyValue<string, bool>("Fan On", false);  Console.WriteLine(kv.key + " is " + kv.val + " years old.");  Console.WriteLine(kv2.key + " is " + kv2.val ); }} |

* Generic structure with constraints: Like generic classes, generic structures can have constraints. Eg: Following *KeyValue* restricts *TValue* to *value* *types*:

**struct** KeyValue<TKey, TValue> **where** TValue : **struct** { /\* . . . \*/

**C#\_9.7 Generic Methods**

As the preceding examples have shown, *methods* inside a *generic class* can make use of a class’ type parameter and are, therefore, *automatically generic* relative to the *type* *parameter*. However, it is possible to *declare a generic method* that uses one or more *type parameters* of its *own*. Furthermore, it is possible to create a *generic method* that is *enclosed* within a *nongeneric* class. Here is the general form of a generic method:

***ret-type meth-name<type-parameter-list>(param-list) {*** *// ...*

In all cases, *type-parameter-list* is a *comma-separated list* of ***type parameters***. Notice that for a generic method, the type parameter list follows the method name.

* The following program declares a non-generic class called ***ArrayUtils*** and a static generic method within that class called ***CopyInsert()***. The ***CopyInsert()*** method copies the contents of *one array to another*, inserting a *new element* at a specified location in the process. It can be used with *any type of array*.

**using System**;

**class** ArrayUtils { *// this is not a generic class.*

**public** **static** **bool** CopyInsert<T>(T e, int idx, T[] src, T[] target) { } *// This is a generic method.*

*/\* . . . \*/* }

**class** GenMethDemo { **static** **void** **Main()** { **int[]** nums = { 1, 2, 3 }; **int[]** nums2 = **new** int[4];

ArrayUtils.CopyInsert(99, 2, nums, nums2); *// Operate on an int array*

*// Now, use CopyInsert on an array of strings.*

**string[]** strs = { "Generics", "are", "powerful."}; **string[]** strs2 = **new** string[4];

ArrayUtils.CopyInsert("in C#", 1, strs, strs2); *// Insert into a string array.*

*// 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);

}}

* First, notice how ***CopyInsert()*** generic method is declared by this line:

***public static bool CopyInsert<T>(T e, int idx, T[] src, T[] target) {***

* The type parameters are declared after the method name, but before the parameter list.
* Also notice that ***CopyInsert()*** is *static*, enabling it to be called *independently* of any *object*. Understand, though, that *generic methods* can be either *static* or *non-static*. There is *no restriction in this regard*.
* type inference: Now, notice how ***CopyInsert()*** is called within ***Main()*** by use of the *normal call syntax*, without the need to specify type arguments. This is because the types of the type arguments are automatically discerned based on the type of data used to call ***CopyInsert()***. Based on this information, the *type* of *T* is *adjusted accordingly*. This process is called type inference.
* In the first call, ***ArrayUtils.CopyInsert(99, 2, nums, nums2);*** the type of T becomes int because 99 is an int, and the element types of nums and nums2 are int.
* In the second call, ***ArrayUtils.CopyInsert("in C#", 1, strs, strs2);*** string types are used, and T is replaced by string.
* Now, notice the *commented-out code*, shown here: ***// ArrayUtils.CopyInsert(0.01, 2, nums, nums2);***
* If you remove the comment symbols, you will receive a *compile-time--error*. The reason is that the type of the first argument is ***double***, but the element types of ***nums*** and ***nums2*** are ***int***.
* All three types must be substituted for the same type parameter, T. Otherwise a type-mismatch occurs, which results in a compile-time error. It ensures type safety for generic methods.
* Using Explicit Type Arguments to Call a Generic Method: Although *implicit type inference* is adequate for most invocations of a *generic method*, it is possible to *explicitly* specify the *type argument*. To do so, specify *the type argument after the method name* when calling the method. Eg: here ***CopyInsert()*** is explicitly specified as type ***string***: ***ArrayUtils.CopyInsert<string>("in C#", 1, strs, strs2);***
* You will need to *explicitly specify* the *type* when the *compiler cannot infer* the type of a *type parameter*.
* Using a Constraint with a Generic Method: You can add constraints to the type arguments of a generic method by *specifying them after the parameter list*. For example, the following version of ***CopyInsert()*** will work only with *reference types*:

***public static bool CopyInsert<T>(T e, int idx, T[] src, T[] target) where T : class {***

* compiler type inference issue with generic method: There are cases in which the compiler cannot infer the *type* to use for a *type parameter* when a *generic method* is called and the *type* will need to be *explicitly specified*. Among others, this situation can occur when a *generic method* *has no parameters*. For example, consider this generic method: ***class SomeClass{ public static T SomeMeth<T>() where T: new(){ return new T(); }*** */\*. . .\*/* ***}***
* When this method is invoked, there are no arguments from which the type of T can be inferred. The return type of T is not sufficient for the inference to take place. Therefore, this won’t work: ***someObj = SomeClass.SomeMeth(); // won't work***
* Instead, it must be invoked with an explicit type specified. For example: ***someObj = SomeClass.SomeMeth<MyClass>(); // fixed***

**C#\_9.8 Generic Delegates**

Like *methods*, delegates can also be generic. To declare a generic delegate, use this general form:

***delegate ret-type delegate-name<type-parameter-list>(arg-list);***

* Notice the *placement* of the type parameter list. It *immediately follows* the delegate’s name.
* The following program demonstrates a generic delegate called ***Invert*** that has one type parameter called ***T***. It returns type T and takes an argument of type T.

|  |  |
| --- | --- |
| delegate T Invert<T>(T v); *// Declare a generic delegate.*  class GenDelegateDemo {  *// Return the reciprocal of a double.*  static double Recip(double v) { return 1 / v; }  *// Reverse a string and return the result.*  static string ReverseStr(string str) { string result = "";  foreach(char ch in str) result = ch + result; return result;} | static void Main() {  *// Construct two Invert delegates.*  Invert<double> invDel = Recip;  Invert<string> invDel2 = ReverseStr;  Console.WriteLine("The reciprocal of 4 is " + invDel(4.0));  Console.WriteLine();  Console.WriteLine("Reversed ABCDEFG: " + invDel2("ABCDEFG ")); }} |

* Don’t get crazy to figure out how the reversing works. The technique is simple- it is just the order of ***char*** and ***result*** in the *assignment expression*:

***result = ch + result;*** If we use ***result = result + ch;*** no reversing will occur.

* ***delegate T Invert<T>(T v);*** Notice that *T* can be used as the *return type* even though the *type* *parameter T* is specified after the name *Invert*.
* Inside ***Main()***, a delegate called ***invDel*** is *instantiated* and assigned a *reference* to ***Recip()***. ***Invert<double> invDel = Recip;***
* Because Recip( ) takes a *double* argument and returns a *double* value, *Recip( )* is compatible with a ***double instance*** of *Invert*.
* Similarly, the delegate called ***invDel2*** is created and assigned a reference to ***ReverseStr()***. ***Invert<string> invDel2 = ReverseStr;***
* Because *ReverseStr( )* takes a *string argument* and returns a *string result*, it is compatible with the *string* *version* of *Invert*.
* Because of the *type-safety* *inherent* in *generics*, you cannot assign incompatible methods to delegates. For example, assuming the preceding program, this statement would be in error: ***Invert<int> invDel = ReverseStr;*** *// Error!*
* Because ***ReverseStr()*** takes a ***string*** *argument* and returns a ***string*** *result*, it cannot be assigned to an ***int*** version of ***Invert***.

**C#\_9.9 Generic Interfaces**

Generic interfaces are specified just like generic classes. Here is an example. It creates a generic interface called ***ITwoDCoord*** that defines methods that get and set X and Y *coordinate values*. Therefore, any class that implements this interface will support X and Y coordinates. The data type of the *coordinates* is specified by a type parameter. ***ITwoDCoord*** is then implemented by ***two different classes***.

|  |  |
| --- | --- |
| *// Demonstrate a generic interface.*  using System;  */\* This interface is generic. It defines methods that support two-dimensional coordinates \*/*  public interface ITwoDCoord<T> { T GetX(); void SetX(T x);  T GetY(); void SetY(T y); } | class XYCoord<T> : ITwoDCoord<T> { T X; T Y; *// A class that encapsulates two-dimensional coordinates.*  public XYCoord(T x, T y) { X = x; Y = y; }  public T GetX() { return X; }  public void SetX(T x) { X = x; }  public T GetY() { return X; }  public void SetY(T y) { Y = y; } } |
| *// A class that encapsulates three-dimensional coordinates.*  class XYZCoord<T> : ITwoDCoord<T> {  T X; T Y; T Z;  public XYZCoord(T x, T y, T z) { X = x; Y = y; Z = z; }  public T GetX() { return X; }  public void SetX(T x) { X = x; }  public T GetY() { return Y; }  public void SetY(T y) { Y = y; }  public T GetZ() { return Z; }  public void SetZ(T z) { Z = z; } } | class GenInterfaceDemo {  */\* A generic method that can display the X,Y coordinates associated with any object that implements the generic interface ITwoDCoord. \*/*  static void ShowXY<T>(ITwoDCoord<T> xy) { Console.WriteLine(xy.GetX() + ", " + xy.GetY()); }  static void Main() {  XYCoord<int> xyObj = new XYCoord<int>(10, 20);  Console.Write("The X,Y values in xyObj: "); ShowXY(xyObj);  XYZCoord<double> xyzObj = new XYZCoord<double>(-1.1, 2.2, 3.1416);  Console.Write("The X,Y component of xyzObj: "); ShowXY(xyzObj); }} |

* Notice how ***ITwoDCoord*** is declared: **public interface ITwoDCoord<T> {**  a *generic interface* uses a *syntax similar* to that of a *generic class*.
* Notice how ***XYCoord***, which implements ***ITwoDCoord***, is declared: ***class XYCoord<T> : ITwoDCoord<T> {***
* The *type parameter* ***T*** is specified by ***XYCoord*** and is also specified in ***ITwoDCoord***.
* IMPORTANT Note: A class that implements a generic version of a generic interface must, itself, be generic. For example, this declaration would be illegal because T is *not defined*: ***class XYCoord : ITwoDCoord<T> {*** *// Wrong!*
* The type parameter T required by ***ITwoDCoord*** must be specified by the implementing class, which is ***XYCoord*** in this case. Otherwise, there is no way for the interface to receive the type argument.
* In ***GenInterfaceDemo***, a generic method called ***ShowXY()*** is defined. It displays the ***X,Y*** *coordinates of the object* that it is passed. Notice that the type of its *parameter* is ITwoDCoord. This means that it can *operate* on any *object* that implements the *ITwoDCoord* *interface*. In this case, it means that objects of type *XYCoord* and *XYZCoord* *can be used* as *arguments*. This fact is illustrated by Main( ).
* Generic interface with constraints: A *type parameter* for a *generic interface* can have *constraints* in the same way as it can for a *generic class*. For example, this version of ***ITwoDCoord*** restricts its use to value types: ***public interface ITwoDCoord<T> where T : struct {***
* When this version is implemented, the implementing class must also specify the same constraint for **T**, as shown here:

***class XYCoord<T> : ITwoDCoord<T> where T : struct {***

Because of the value type constraint, this version of XYCoord cannot be used on class types, for example. Thus, the *following declaration* would be disallowed:

***XYCoord<string> xyObj = new XYCoord<string>("10", "20");****// Now, this won't work.*

* Because string is not a value type, its use with XYCoord is illegal.
* Although a class that implements a generic version of a *generic interface* must, itself, be *generic*, as explained earlier, a *non-generic class* *can implement a specific version* of a generic interface. For example, here, ***XYCoordInt*** *explicitly implements* ***ITwoDCoord<int>*** :

|  |  |
| --- | --- |
| **class** XYCoordInt : ITwoDCoord<**int**> {  **int** X; **int** Y;  **public** XYCoordInt(**int** x, **int** y) { X = x; Y = y; }  **public** **int** GetX() { **return** X; }  **public** **void** SetX(**int** x) { X = x; }  **public** **int** GetY() { **return** X; }  **public** **void** SetY(**int** y) { Y = y; } } | * Notice that ***ITwoDCoord*** is specified with an explicit ***int*** type. Therefore, ***XYCoordInt*** *does not need to take a type argument* because it does not pass it along to ***ITwoDCoord***. * Although a *property* *declaration* cannot, itself, specify a *type parameter*, a ***property*** *declared in a* ***generic class*** *can use a* ***type parameter***that is declared by the ***generic class***. Therefore, the methods ***GetX()***, ***GetY()***, and so on in the preceding example can be made into ***properties*** that use the ***type parameter*** ***T***. |

**C#\_9.10 Comparing two type parameters using the = = or ! = operators**

If the type parameter specifies a reference or a base class constraint, then = = and ! = are allowed, but they *only test for* *reference equ*ality. For example, this method will not compile: **public** **static** **bool** SameValue<T>(T a, T b) { **if**(a == b) **return** **true**; *// Won't work*

**return** **false**; }

Because T is a generic type, the compiler has no way to know precisely how two objects should be compared for equality. *Should a bitwise comparison be done? Should only certain fields be compared? Should reference equality be used? The compiler has no way to answer these questions.* At first glance, this seems to be a serious problem. Fortunately, it isn’t because C# providesa mechanism by which you can *determine* if two instances of a type parameter are the same.

* IComparable: To enable two objects of a generic type parameter to be compared, use the ***CompareTo()*** method defined by one of the standard interfaces: ***IComparable***.
* This interface has both a generic and a non-generic form. ***IComparable*** is implemented by all of C#’s built-in types, including ***int***, ***string***, and ***double***. It is also easy to *implement for classes* that you create.
* The ***IComparable*** interface defines only the ***CompareTo()*** method. Its generic form is: ***int CompareTo(T obj)***
* It compares the invoking object to obj. It returns zero if the two objects are *equal*, a positive value if the *invoking object is greater* than *obj*, and a negative value if the *invoking object* *is less* than *obj*.
* To use ***CompareTo()***, you must specify a constraint that requires every type argument to implement the ***IComparable*** interface. Then, when you need to compare two objects of the type parameter, simply call ***CompareTo()***. For example, here is a corrected version of ***SameValue()***:

*// Require IComparable interface.*

**public static bool** SameValue<T>(T a, T b) **where** T : IComparable<T> { **if**(a.CompareTo(b) == 0) **return** **true**; *// fixed*

**return** **false**; }

* Because the interface constraint requires that ***T*** implement ***IComparable<T>***, the ***CompareTo()*** method can be used to *determine equality*. Of course, this means that *the only instances of classes that implement* ***IComparable<T>*** can be passed to ***SameValue()***.