

Generic Classes (C# Programming Guide)

Article • 07/09/2022

Generic classes encapsulate operations that are not specific to a particular data type. The most common use for generic classes is with collections like linked lists, hash tables, stacks, queues, trees, and so on. Operations such as adding and removing items from the collection are performed in basically the same way regardless of the type of data being stored.

For most scenarios that require collection classes, the recommended approach is to use the ones provided in the .NET class library. For more information about using these classes, see [Generic Collections in .NET](#).

Typically, you create generic classes by starting with an existing concrete class, and changing types into type parameters one at a time until you reach the optimal balance of generalization and usability. When creating your own generic classes, important considerations include the following:

- Which types to generalize into type parameters.

As a rule, the more types you can parameterize, the more flexible and reusable your code becomes. However, too much generalization can create code that is difficult for other developers to read or understand.

- What constraints, if any, to apply to the type parameters (See [Constraints on Type Parameters](#)).

A good rule is to apply the maximum constraints possible that will still let you handle the types you must handle. For example, if you know that your generic class is intended for use only with reference types, apply the class constraint. That will prevent unintended use of your class with value types, and will enable you to use the `as` operator on `T`, and check for null values.

- Whether to factor generic behavior into base classes and subclasses.

Because generic classes can serve as base classes, the same design considerations apply here as with non-generic classes. See the rules about inheriting from generic base classes later in this topic.

- Whether to implement one or more generic interfaces.

For example, if you are designing a class that will be used to create items in a generics-based collection, you may have to implement an interface such as `IComparable<T>` where `T` is the type of your class.

For an example of a simple generic class, see [Introduction to Generics](#).

The rules for type parameters and constraints have several implications for generic class behavior, especially regarding inheritance and member accessibility. Before proceeding, you should understand some terms. For a generic class `Node<T>`, client code can reference the class either by specifying a type argument - to create a closed constructed type (`Node<int>`); or by leaving the type parameter unspecified - for example when you specify a generic base class, to create an open constructed type (`Node<T>`). Generic classes can inherit from concrete, closed constructed, or open constructed base classes:

C#

```
class BaseNode { }
class BaseNodeGeneric<T> { }

// concrete type
class NodeConcrete<T> : BaseNode { }

//closed constructed type
class NodeClosed<T> : BaseNodeGeneric<int> { }

//open constructed type
class NodeOpen<T> : BaseNodeGeneric<T> { }
```

Non-generic, in other words, concrete, classes can inherit from closed constructed base classes, but not from open constructed classes or from type parameters because there is no way at run time for client code to supply the type argument required to instantiate the base class.

C#

```
//No error
class Node1 : BaseNodeGeneric<int> { }

//Generates an error
//class Node2 : BaseNodeGeneric<T> {}

//Generates an error
//class Node3 : T {}
```

Generic classes that inherit from open constructed types must supply type arguments for any base class type parameters that are not shared by the inheriting class, as

demonstrated in the following code:

C#

```
class BaseNodeMultiple<T, U> { }

//No error
class Node4<T> : BaseNodeMultiple<T, int> { }

//No error
class Node5<T, U> : BaseNodeMultiple<T, U> { }

//Generates an error
//class Node6<T> : BaseNodeMultiple<T, U> { }
```

Generic classes that inherit from open constructed types must specify constraints that are a superset of, or imply, the constraints on the base type:

C#

```
class NodeItem<T> where T : System.IComparable<T>, new() { }
class SpecialNodeItem<T> : NodeItem<T> where T :
System.IComparable<T>, new() { }
```

Generic types can use multiple type parameters and constraints, as follows:

C#

```
class SuperKeyType<K, V, U>
    where U : System.IComparable<U>
    where V : new()
{ }
```

Open constructed and closed constructed types can be used as method parameters:

C#

```
void Swap<T>(List<T> list1, List<T> list2)
{
    //code to swap items
}

void Swap(List<int> list1, List<int> list2)
{
    //code to swap items
}
```

If a generic class implements an interface, all instances of that class can be cast to that interface.

Generic classes are invariant. In other words, if an input parameter specifies a `List<BaseClass>`, you will get a compile-time error if you try to provide a `List<DerivedClass>`.

See also

- [System.Collections.Generic](#)
- [C# Programming Guide](#)
- [Generics](#)
- [Saving the State of Enumerators](#)
- [An Inheritance Puzzle, Part One](#)