♦ Generics

- **Functionally similar to C++ Templates**
  - JIT compiler duplicates class structure for different types
- **Why use Generics**
  - Performance (no need for *boxing* and *unboxing*)
  - Type Safety
- **Generics in the .NET Framework Class Library**
  - System.Collections.Generic
- **Create a Custom Generic Class**
- **Constraints on Type Parameters**

Generic classes (or **generics**) are functionally similar to **templates in C++**. They aren't really a new invention of Microsoft, but were introduced to C# 2.0 in 2005. The idea behind generics is to develop universal classes with a special syntax to accept a type as a parameter, so that all code in the universal classes works with the special parameter. At **runtime** the Just-in-Time (JIT) compiler creates a separate copy of the class for every type specified while using the generic class. Hence the **JIT compiler**, rather than the developer, **does the work to duplicate the class structure for different types**.

In other words: generics is the implementation of *parametric polymorphism*. Using parametric polymorphism, a method or data type can be written generically so that it can deal equally well with objects of various types. It is a way to make a language more expressible, while still maintaining full static type-safety.

Why use Generics

There are mainly two reasons to use generics. These are:

- *Performance* – Collections that store objects use *Boxing* and *Unboxing* on data types. This uses a significant amount of overhead, which can give a performance hit. By using generics instead, this performance hit is removed.
- *Type Safety* – Using generics allows the *compiler* to catch this type of bug before your program runs. Additionally, you can specify constraints to limit the classes used in a generic, enabling the compiler to detect an incompatible type (see section “Constraints on Type Parameters”).

Example of Using Generics

Consider the following class, representing games for a certain handheld game console:

```
class Game
{
    private string name;
    private int minimumAge;

    // Parameterless public empty (default) constructor.
    public Game() { }

    public Game(string Name, int MinimumAge)
    {
        name = Name;
        minimumAge = MinimumAge;
```

```
    }

    public string Name
    {
        get { return name; }
        set { name = value; }
    }

    public int MinimumAge
    {
        get { return minimumAge; }
        set { minimumAge = value; }
    }

    public override string ToString()
    {
        return string.Format("title '{0}' is rated {1}+", name, minimumAge);
    }
}
```

Without generics, you could use an ArrayList to store several Game instances:

```
ArrayList listOfGames = new ArrayList();

listOfGames.Add(new Game("Open Season", 3));
listOfGames.Add(new Game("Prince of Persia", 16));
listOfGames.Add(new Game("Peter Jackson's King Kong", 12));

// This line can be compiled without a problem...
listOfGames.Add("Hello");

// ... but will cause the foreach loop to break at runtime -> System.InvalidCastException
foreach(Game g in ListOfGames)
{
    Console.WriteLine(g);
}
```

As mentioned in the commented lines the error will not be detected until run-time. This can be avoided by using the following code instead:

```
// Declaration and instantiation of the actual List generic type.
List<Game> genericListOfGames = new List<Game>();

genericListOfGames.Add(new Game("Grand Theft Auto - Liberty City Stories", 18));
genericListOfGames.Add(new Game("James Bond - From Russia With Love", 16));
genericListOfGames.Add(new Game("Lemmings", 3));

// Compile error: The best overloaded method match for
// 'System.Collections.Generic.List<Generics.Game>
// .Add(Generics.Game)' has some invalid arguments
genericListOfGames.Add("Hello");

foreach(Game g in genericListOfGames)
{
    Console.WriteLine(g);
}
```

This last piece of code will not even build, because at the declaration and instantiation of the List type the compiler is told that only objects of the Game type (and any of its *descendants* if they existed) can be stored in the List object.

Generics in the .NET Framework Class Library

Since version 2.0 the .NET Framework class library provides a new namespace, *System.Collections.Generic*, which includes several ready-to-use generic collection classes and associated interfaces. Before designing and implementing your own custom collection classes, consider whether you can use or derive a class from one of the classes provided in the base class library.

As you can see in the following table, most collection classes known since Framework 1.x have a generic equivalent:

Non-Generic Class	Generic Equivalent
ArrayList	List<>
DictionaryEntry	KeyValuePair<>
Hashtable	Dictionary<>
HybridDictionary	Dictionary<>
ListDictionary	Dictionary<>
NameValueCollection	Dictionary<>
N/A	SortedDictionary<>
N/A	LinkedList<>
OrderedDictionary	Dictionary<>
Queue	Queue<>
SortedList	SortedList<>
Stack	Stack<>
StringCollection	List<String>
StringDictionary	Dictionary<String>

Our game example could also be coded as follows, using a Dictionary generic type:

```
// Declaration and instantiation of a Dictionary generic type.
Dictionary<int, Game> dictionaryOfGames = new Dictionary<int, Game>();

dictionaryOfGames.Add(31, new Game("Open Season", 3));
dictionaryOfGames.Add(12, new Game("Prince of Persia", 16));

// Another way to fill a Dictionary element.
dictionaryOfGames[42] = new Game("King Kong", 12);

foreach(KeyValuePair<int, Game> g in dictionaryOfGames)
{
    Console.WriteLine("element with key {0} has rating {1}+", g.Key, g.Value.MinimumAge);
}
```

Create a Custom Generic Class

If none of the generic collection classes provided by the .NET Framework satisfy your programming desires, a custom generic collection class can be made.

In our game example, let's assume it is required that the collection class can be queried about the lowest and highest minimum age of all the games it contains. Since this is not standard functionality, we have to create our own generic class:

```
// For an explanation of the where clause,
// see section "Constraints on Type Parameters".
class CustomGameCollection<T> : IEnumerable<T> where T : class, new()
{
    private List<T> gameList = new List<T>();

    private int lowestAge  = 0;
    private int highestAge = 0;

    public T GetThing(int position)
    {
        return gameList[position];
    }

    public void AddThing(T item)
    {
        gameList.Add(item);

        if (item is Game)
        {
            int itemAge = (item as Game).MinimumAge;

            lowestAge  = Math.Min(lowestAge,  itemAge);
            highestAge = Math.Max(highestAge, itemAge);

            if (lowestAge == 0) lowestAge = itemAge;
        }
```

```
    }

    public int GetLowestMinAge()
    {
        return lowestAge;
    }

    public int GetHighestMinAge()
    {
        return highestAge;
    }

    // IEnumerable<T> extends IEnumerable, therefore we
    // need to implement both versions of GetEnumerator().
    public IEnumerator<T> GetEnumerator()
    {
        return gameList.GetEnumerator();
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return gameList.GetEnumerator();
    }
}
```

The code using our custom generic class could look as follows:

```
CustomGameCollection<Game> cgc = new CustomGameCollection<Game>();

cgc.AddThing(new Game("Loco Roco", 3));
cgc.AddThing(new Game("Test Drive Unlimited", 3));
cgc.AddThing(new Game("SOCOM US Navy Seals Fireteam Bravo", 17));

foreach (Game g in cgc)
{
    Console.WriteLine(g);
}

lstGames.Items.Add("Lowest Minimum Age: {0}, Highest Minimum Age: {1}",
    cgc.GetLowestMinAge(), cgc.GetHighestMinAge()));
```

Constraints on Type Parameters

Constraints are used to restrict a generic class to the kinds of types that client code can use for type arguments. If code attempts to instantiate the class with a type that is not allowed by a constraint, then this will result in a *compile-time* error. Constraints are specified using the **where** keyword. The following table lists the types of constraints:

Constraint	Description
`where T : struct`	Type argument T must be any value type except Nullable.
`where T : class`	Type argument T must be a reference type (any class, interface, delegate or array type).
`where T : new()`	Type argument T must have a public no-argument constructor. When used in conjunction with other constraints, the new() constraint must be specified last.
`where T : <base class name>`	Type argument T must be or derive from the specified base class.
`where T : <interface name>`	Type argument T must be or implement the specified interface. Multiple interface constraints can be specified. The constraining interface can also be generic.

Note For more info on constraints on type parameters, see *http://msdn2.microsoft.com/en-us/library/d5x73970.aspx*