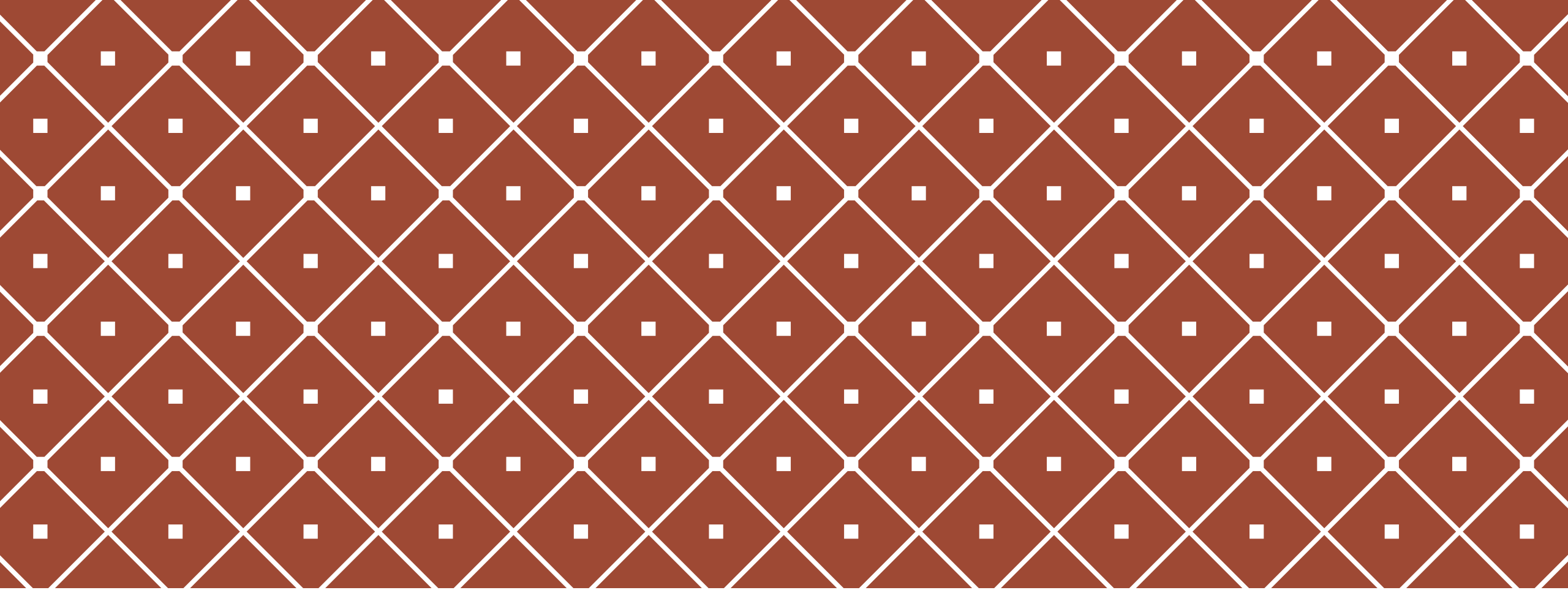

GENERICS AND C#

Elizabeth Bourke

INTRODUCTION TO GENERICS

In any software project, bugs are simply a fact of life. Careful planning, programming, and testing can help reduce them. This becomes especially apparent as new features are introduced and your code base grows in size and complexity.

Fortunately, some bugs are easier to detect than others. Compile-time bugs, for example, can be detected early on; you can use the compiler's error messages to figure out what the problem is and fix it, right then and there.

Runtime bugs, however, can be much more problematic; they don't always surface immediately, and when they do, it may be at a point in the program that is far removed from the actual cause of the problem.

When you use generics, the compiler makes compile time checks on your code for conformance to type safety.

EXAMPLE

ArrayList is a highly convenient collection class that can be used without modification to store any reference or value type.

```
// The .NET Framework 1.1 way to create a list:
System.Collections.ArrayList list1 = new System.Collections.ArrayList();
list1.Add(3);
list1.Add(105);

System.Collections.ArrayList list2 = new System.Collections.ArrayList();
list2.Add("It is raining in Redmond.");
list2.Add("It is snowing in the mountains.");
```

NON-GENERIC CODE

But this convenience comes at a cost.

Any reference or value type that is added to an ArrayList is implicitly upcast to Object.

If the items are value types, they must be boxed when they are added to the list, and unboxed when they are retrieved.

Both the casting and the boxing and unboxing operations decrease performance;

the effect of boxing and unboxing can be very significant in scenarios where you must iterate over large collections.

NON-GENERIC CODE

The other limitation is lack of compile-time type checking; because an ArrayList casts everything to Object, there is no way at compile-time to prevent client code from doing something such as this:

```
System.Collections.ArrayList list = new System.Collections.ArrayList();
// Add an integer to the list.
list.Add(3);
// Add a string to the list. This will compile, but may cause an error later.
list.Add("It is raining in Redmond.");

int t = 0;
// This causes an InvalidCastException to be returned.
foreach (int x in list)
{
    t += x;
}
```

NON-GENERIC CODE

In versions 1.0 and 1.1 of the C# language, you could avoid the dangers of generalized code in the .NET Framework base class library collection classes only by writing your own type specific collections.

Of course, because such a class is not reusable for more than one data type, you lose the benefits of generalization, and you have to rewrite the class for each type that will be stored.

What ArrayList and other similar classes really need is a way for client code to specify, on a per-instance basis, the particular data type that they intend to use. That would eliminate the need for the upcast and would also make it possible for the compiler to do type checking.

In other words, ArrayList needs a type parameter. That is exactly what generics provide.

GENERIC EXAMPLE

In the generic List<T> collection, in the **System.Collections.Generic** namespace, the same operation of adding items to the collection resembles this:

```
// The .NET Framework 2.0 way to create a list
List<int> list1 = new List<int>();

// No boxing, no casting:
list1.Add(3);

// Compile-time error:
// list1.Add("It is raining in Redmond.");
```

GENERICS

In a nutshell, generics enable types (classes and interfaces) to be parameters when defining classes, interfaces and methods.

This implies that when using a strongly types language you should declare a type prior to storing data in it.

Generics enable you to work with typesafe data with no boxing and unboxing overhead.

When you use generics, the compiler makes compile time checks on your code for conformance to type safety. Code that makes use of generics is always due to the avoidance of boxing and unboxing overheads.

GENERICS EXAMPLE

Generics help you implement algorithms that can work on a wide variety of types. As an example, you may want to write an algorithm to sort an array of integers or doubles or even an array of strings. To implement such sorting algorithms without generics you would typically need multiple sort methods -- one per each type.

```
public static int[] Sort(int[] integerArray)
{
    //Code to sort an array of integers
    return integerArray;
}
```

```
public double[] Sort(double[] doubleArray)
{
    //Code to sort an array of doubles
    return doubleArray;
}
```

```
public static string[] Sort(string[] stringArray)
{
    //Code to sort an array of strings
    return stringArray;
}
```

GENERICS EXAMPLE

If you use Generics, you can just have one method that can accept a type parameter and sort the input data when asked.

```
public class Algorithm<T>
    {
        public static T[] Sort(T[] inputArray)
        {
            //Code to sort a generic array
            return inputArray;
        }
    }
```

GENERIC COLLECTIONS

Generic classes and methods combine reusability, type safety and efficiency in a way that their non-generic counterparts cannot.

Generics are most frequently used with collections and the methods that operate on them.

Version 2.0 of the .NET Framework class library provides a new namespace, System.Collections.Generic, which contains several new generic-based collection classes.

It is recommended that all applications that target the .NET Framework 2.0 and later use the new generic collection classes instead of the older non-generic counterparts such as ArrayList

GENERIC COLLECTIONS

The System.Collections.Generic namespace contains interfaces and classes that define generic collections.

List<T> Class

```
List<int> myInts = new List<int>();

myInts.Add(1);
myInts.Add(2);
myInts.Add(3);

for (int i = 0; i < myInts.Count; i++)
{
    Console.WriteLine("MyInts: {0}", myInts[i]);
}
```

Dictionary<TKey, TValue> Collection

Another very useful generic collection is the *Dictionary*, which works with key/value pairs.

There is a non-generic collection, called a *Hashtable* that does the same thing, except that it operates on type *object*.

However, you will want to avoid the non-generic collections and use the generic counterparts instead.

Example: a list of *Customers* that you need to work with. It would be natural to keep track of these *Customers* via their *CustomerID*. The *Dictionary* example will work with instances of the following *Customer* class:

DICTIONARY EXAMPLE

```
public class Customer
{
    public Customer(int id, string name)
    {
        ID = id;
        Name = name;
    }

    private int m_id;

    public int ID
    {
        get { return m_id; }
        set { m_id = value; }
    }

    private string m_name;

    public string Name
    {
        get { return m_name; }
        set { m_name = value; }
    }
}
```

```
Customer cust1 = new Customer(1, "Cust 1");
Customer cust2 = new Customer(2, "Cust 2");
Customer cust3 = new Customer(3, "Cust 3");

customers.Add(cust1.ID, cust1);
customers.Add(cust2.ID, cust2);
customers.Add(cust3.ID, cust3);

foreach (KeyValuePair<int, Customer> custKeyVal in customers)
{
    Console.WriteLine(
        "Customer ID: {0}, Name: {1}",
        custKeyVal.Key,
        custKeyVal.Value.Name);
}
```

Iterating through the *customers Dictionary* with a *foreach* loop, the type returned is *KeyValuePair<TKey, TValue>*, where *TKey* is type *int* and *TValue* is type *Customer* because those are the types that the *customers Dictionary* is defined with.
Since *custKeyVal* is type *KeyValuePair<int, Customer>* it has *Key* and *Value* properties for you to read from.
custKeyVal.Key will hold the *ID* for the *Customer* instance and *custKeyVal.Value* will hold the whole *Customer* instance. Dislaying the customer *Name* is obtained through the *Name* property of the *Customer* instance that is returned by the *Value* property.

SORTEDDICTIONARY <TKEY, TVALUE> CLASS

Represents a collection of key/value pairs that are sorted on the key.

```
using System;
using System.Collections.Generic;

public class Example
{
    public static void Main()
    {
        // Create a new sorted dictionary of strings, with string
        // keys.
        SortedDictionary<string, string> openWith =
            new SortedDictionary<string, string>();

        // Add some elements to the dictionary. There are no
        // duplicate keys, but some of the values are duplicates.
        openWith.Add("txt", "notepad.exe");
        openWith.Add("bmp", "paint.exe");
        openWith.Add("dib", "paint.exe");
        openWith.Add("rtf", "wordpad.exe");

        // The Add method throws an exception if the new key is
        // already in the dictionary.
        try
        {
            openWith.Add("txt", "winword.exe");
        }
        catch (ArgumentException)
        {
            Console.WriteLine("An element with Key = \"txt\" already exists.");
        }


        // When you use foreach to enumerate dictionary elements,
        // the elements are retrieved as KeyValuePair objects.
        Console.WriteLine();
        foreach( KeyValuePair<string, string> kvp in openWith )
        {
            Console.WriteLine("Key = {0}, Value = {1}",
                kvp.Key, kvp.Value);
        }
```

```
An element with Key = "txt" already exists.

Key = bmp, Value = paint.exe
Key = dib, Value = paint.exe
Key = doc, Value = winword.exe
Key = ht, Value = hypertrm.exe
Key = rtf, Value = winword.exe
Key = txt, Value = notepad.exe
```

HASHSET<T> CLASS

Represents a set of values.

The HashSet<T> class provides high-performance set operations. A set is a collection that contains no duplicate elements, and whose elements are in no particular order.

SortedSet<T> Class
Represents a collection of objects that is maintained in sorted order.

```
using System;
using System.Collections.Generic;

class Program
{
    static void Main()
    {
        HashSet<int> evenNumbers = new HashSet<int>();
        HashSet<int> oddNumbers = new HashSet<int>();

        for (int i = 0; i < 5; i++)
        {
            // Populate numbers with just even numbers.
            evenNumbers.Add(i * 2);

            // Populate oddNumbers with just odd numbers.
            oddNumbers.Add((i * 2) + 1);
        }

        Console.Write("evenNumbers contains {0} elements: ", evenNumbers.Count);
        DisplaySet(evenNumbers);

        Console.Write("oddNumbers contains {0} elements: ", oddNumbers.Count);
        DisplaySet(oddNumbers);

        // Create a new HashSet populated with even numbers.
        HashSet<int> numbers = new HashSet<int>(evenNumbers);
        Console.WriteLine("numbers UnionWith oddNumbers...");
        numbers.UnionWith(oddNumbers);

        Console.Write("numbers contains {0} elements: ", numbers.Count);
        DisplaySet(numbers);

    }

    private static void DisplaySet(HashSet<int> set)
    {
        Console.Write("{");
        foreach (int i in set)
        {
            Console.Write(" {0}", i);
        }
        Console.WriteLine(" }");
    }
}
/* This example produces output similar to the following:
 * evenNumbers contains 5 elements: { 0 2 4 6 8 }
 * oddNumbers contains 5 elements: { 1 3 5 7 9 }
 * numbers UnionWith oddNumbers...
 * numbers contains 10 elements: { 0 2 4 6 8 1 3 5 7 9 }
 */
```

WHY USE GENERICS?

Code that uses generics has many benefits over non-generic code:

Stronger type checks at compile time. A compiler applies strong type checking to generic code and issues errors if the code violates type safety. Fixing compile-time errors is easier than fixing runtime errors, which can be difficult to find. Generics add stability to your code by making more of your bugs detectable at compile time.

Build re-usable code with generics.

Casting, boxing unboxing removed – increases efficiency

Less code to maintaine