Boxing, Unboxing, and LINQ

Doug Blank
Bryn Mawr College
Programming Paradigms
Spring 2010

Value and Object Types in C#

```
// Value type:
int i = 5;

// Object/Reference type:
Point p = new Point(1, 34);

// Any object/reference type:
Object obj = p;

// Casting back to original type:
((Point) obj).x
```

Values and Reference Types

- Value access is fast when you know you are dealing with primitive types
- Values use less memory for these primitive types
- Objects use more memory, but are necessary
- Only passes a “reference” to the object, not a copy of an object
- Value types are stored on “the stack”
- Reference types are stored on “the heap”

Problem!

// What about:
Object obj = 5;

One could require every type to be an object, but that would use up a lot of memory, and would require time to create the objects, and might not ever be used.

What is the solution?

Unified Type System

To be able to deal generically with:

Object o = thing;

without worrying whether thing is a type or reference, C# introduces the ideas of **boxing** and **unboxing**

Boxing and Unboxing

```
class Test
{
    static void Main() {
    int i = 1;
    Object o = i; // automatic boxing
    int j = (int) o; // unboxing
    }
}
```

Generically dealing with Objects

```
public static void UseArrayList( )
{
   // Create and populate an ArrayList.
   ArrayList numbers = new ArrayList( );
   numbers.Add(1); // Causes a boxing operation to occur
   numbers.Add(2); // Causes a boxing operation to occur

   // Display all integers in the ArrayList.
   // Causes an unboxing operation to occur on each iteration
   foreach (int i in numbers)
   {
      Console.WriteLine(i);
   }
   numbers.Clear( );
}
```

Generics

```
// Create and populate a List.
   List<int> numbers = new List<int>( );
   numbers.Add(1);
   numbers.Add(2);

// Create and populate a List.
   List<string> strings = new List<string>( );
   numbers.Add(“hello”);
   numbers.Add(“world”);
```

Generic Collections

```
using System.Collections.Generic;


  List<string> people = new List<string>()
  {
     "Granville", "John", "Rachel", "Betty",
    "Chandler", "Ross", "Monica"
  };
```

LINQ

- Language Integrated Query
- Allows more flexibly written code
- Allows system to find better optimizations then that written by hand

Before LINQ

```
int someValue = 5;

var results =
    SomeCollection
      .Where(c => c.SomeProperty < someValue * 2)
      .Select(c => new List<object>() {c.SomeProperty,
                                       c.OtherProperty});

foreach (var result in results)
{
    Console.WriteLine(result.ToString());
}
```

Method-based Queries

```
results = SomeCollection.Where(...).Select(...);
```

Structured Query Language

- Quel – Query Language
- SQL - “sequel” (joke---came after Quel)
- select name, age from people where age > 18;
- Database is free to figure out the best way to interpret the query
- C# version does not give the system this freedom

LINQ

```
int someValue = 5;

var results =
        from c in SomeCollection
          where c.SomeProperty < someValue * 2
              select new {c.SomeProperty,
                          c.OtherProperty};

foreach (var result in results)
{
        Console.WriteLine(result);
}
```

LINQ

- New syntax (syntactic sugar)
- Allows query to be a “first-class object” in language
- Some shortcuts
  - Implicitly-typed variables
  - Anonymous types
  - Lambda expressions

LINQ

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

class Example {

  private static List<string> people = new List<string>()
  {
    "Rachel", "Betty", "Chandler", "Ross", "Monica"
  };

  public static void Main()
  {
    IEnumerable<string> query = from p in people select p;
    foreach (string person in query)
    {
      Console.WriteLine(person);
    }
  }
}
```

LINQ

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

class Example {

  public static void Main()
  {
    foreach (string person in from p in new List<string>()
                { "Rachel", "Betty", "Chandler", "Ross", "Monica" }
                select p)
      {
       Console.WriteLine(person);
      }
  }
}
```