Generics

Often enough the need arises to create a class or method that can work equally well with any data type. Polymorphism and inheritance can help only to a certain extent; a method that requires complete genericity of its parameters is forced to work with `System.Object`, which leads to the two primary problems of generic programming in .NET prior to .NET 2.0:

- *Type safety*: how to verify operations on generic data types at compile-time and explicitly prohibit anything that may fail at runtime?
- *No boxing*: how to avoid boxing value types when the method's parameters are `System.Object` references?

These are no small problems. To see why, consider the implementation of one of the simplest collections in .NET 1.1, the `ArrayList`. The following is a trivialized implementation that nonetheless exhibits the problems alluded above:

```
public class ArrayList : IEnumerable, ICollection, IList, ... {
  private object[] items;
  private int size;
  public ArrayList(int initialCapacity) {
   items = new object[initialCapacity];
  }
  public void Add(object item) {
   if (size < items.Length – 1) {
   items[size] = item;
   ++size;
   } else {
   //Allocate a larger array, copy the elements, then place ‘item’ in it
   }
  }
  public object this[int index] {
   get {
   if (index < 0 || index >= size) throw IndexOutOfBoundsException(index);
   return items[index];
   }
   set {
   if (index < 0 || index >= size) throw IndexOutOfBoundsException(index);
   items[index] = value;
   }
  }
  //Many more methods omitted for brevity
}
```

We have highlighted throughout the code all the occurrences of `System.Object`, which is the “generic” type on which the collection is based. Although this may appear to be a perfectly valid solution, actual usage is not quite so flawless:

```
ArrayList employees = new ArrayList(7);
employees.Add(new Employee(“Kate”));
employees.Add(new Employee(“Mike”));
Employee first = (Employee)employees[0];
```

This ugly downcast to `Employee` is required because the `ArrayList` does not retain any information about the type of elements in it. Furthermore, it does not constrain objects inserted into it to have any common traits. Consider:

```
employees.Add(42);                              //Compiles and works at runtime!
Employee third = (Employee)employees[2];        //Compiles and throws an exception at runtime...
```

Indeed, the number 42 does not belong in a collection of employees, but nowhere did we specify that the `ArrayList` is restricted to instances of a particular type. Although it is possible to create an `ArrayList` implementation that restricts its items to a particular type, this would still be a costly runtime operation and a statement like `employees.Add(42)` would not fail to compile.

This is the problem of *type safety*; “generic” collections based on `System.Object` cannot guarantee compile-time type safety and defer all checks (if any) to runtime. However, this could be the least of our concerns from a *performance* viewpoint — but it turns out that there is a serious performance problem where value types are involved. Examine the following code, which uses the `Point2D` struct from Chapter 3 (a simple value type with X and Y integer coordinates):

```
ArrayList line = new ArrayList(1000000);
for (int i = 0; i < 1000000; ++i) {
  line.Add(new Point2D(i, i));
}
```

Every instance of `Point2D` inserted to the `ArrayList` is boxed, because its `Add` method accepts a reference type parameter (`System.Object`). This incurs the cost of 1,000,000 heap allocations for boxed `Point2D` objects. As we have seen in Chapter 3, on a 32-bit heap 1,000,000 boxed `Point2D` objects would occupy 16,000,000 bytes of memory (compared to 8,000,000 bytes as plain value types). Additionally, the items reference array inside the `ArrayList` would have at least 1,000,000 references, which amounts to another 4,000,000 bytes — a total of 20,000,000 bytes (see Figure 5-1) where 8,000,000 would suffice. Indeed, this is the exact same problem which led us to abandon the idea of making `Point2D` a reference type; `ArrayList` forces upon us a collection that works well only with reference types!

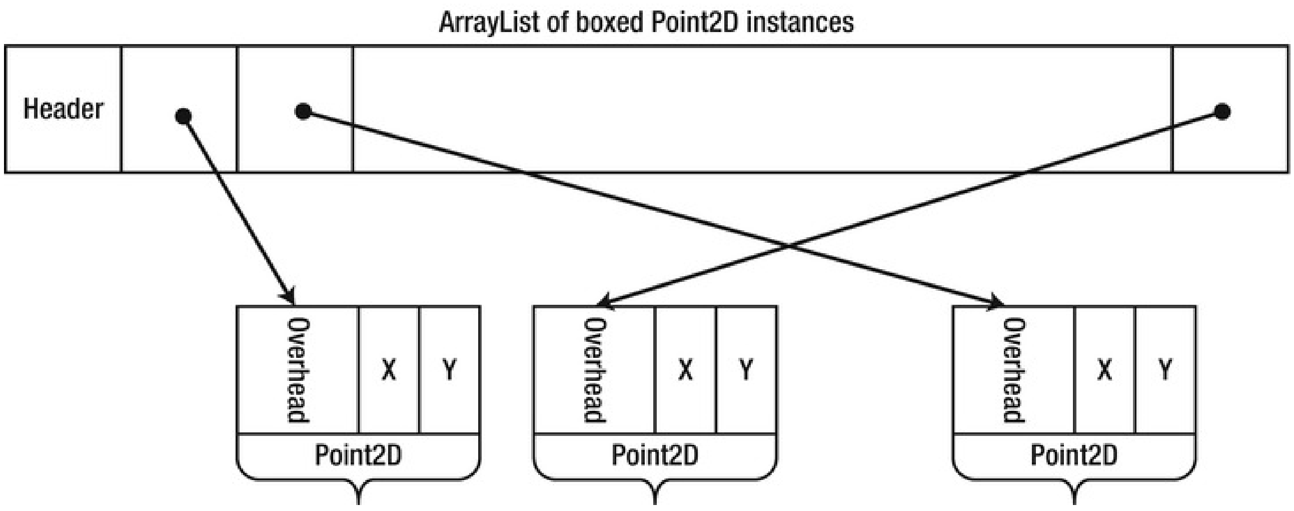

Figure 5-1 . An ArrayList that contains boxed Point2D objects stores references which occupy extra memory and forces the Point2D objects to be boxed on the heap, which adds overhead to them as well

Is there any room for improvement? Indeed, we could write a specialized collection for two-dimensional points, as follows (although we would also have to come up with specialized `IEnumerable`, `ICollection`, and `IList` interfaces for points...). It is exactly identical to the “generic” `ArrayList`, but has `Point2D` wherever we previously had `object`:

```
public class Point2DArrayList : IPoint2DEnumerable, IPoint2DCollection, IPoint2DList, ... {
  private Point2D[] items;
  private int size;
  public ArrayList(int initialCapacity) {
   items = new Point2D[initialCapacity];
  }
  public void Add(Point2D item) {
   if (size < items.Length – 1) {
   items[size] = item;
   ++size;
   } else {
   //Allocate a larger array, copy the elements, then place ‘item’ in it
   }
  }
  public Point2D this[int index] {
   get {
   if (index < 0 || index >= size) throw IndexOutOfBoundsException(index);
   return items[index];
   }
   set {
   if (index < 0 || index >= size) throw IndexOutOfBoundsException(index);
   items[index] = value;
   }
  }
  //Many more methods omitted for brevity
}
```

A similar specialized collection for `Employee` objects would address the type safety problem we have considered earlier. Unfortunately, it is hardly practical to develop a specialized collection for every data type. This is precisely the role of the language compiler as of .NET 2.0 — to allow generic data types in classes and methods while achieving type safety and doing away with boxing.

.NET Generics

Generic classes and methods allow *real* generic code, which does not fall back to `System.Object` on the one hand and does not require specialization for each data type on the other hand. The following is a sketch of a generic type, `List<T>`, which replaces our previous `ArrayList` experiment and addresses both the type safety and boxing concerns:

```
public class List<T> : IEnumerable<T>, ICollection<T>, IList<T>, ... {
  private T[] items;
  private int size;
  public List(int initialCapacity) {
   items = new T[initialCapacity];
  }
  public void Add(T item) {
   if (size < items.Length – 1) {
   items[size] = item;
   ++size;
   } else {
   //Allocate a larger array, copy the elements, then place ‘item’ in it
   }
  }
  public T this[int index] {
   get {
   if (index < 0 || index >= size) throw IndexOutOfBoundsException(index);
   return items[index];
   }
   set {
   if (index < 0 || index >= size) throw IndexOutOfBoundsException(index);
   items[index] = value;
   }
   }
   //Many more methods omitted for brevity
}
```

Note If you are not familiar at all with the syntax of C# generics, Jon Skeet’s “C# in Depth” (Manning, 2010) is a good textbook. Through the rest of this chapter we assume you have written a generic class or at least consumed generic classes, such as the .NET Framework’s collections.

If you have ever written a generic class or method, you know how easy it is to convert pseudo-generic code based on `System.Object` instances to truly generic code, with one or more *generic type parameters*. It is also exceptionally easy to use generic classes and methods by substituting *generic type arguments* where necessary:

```
List<Employee> employees = new List<Employee>(7);
employees.Add(new Employee(“Kate”));
Employee first = employees[0];    //No downcast required, completely type-safe
employees.Add(42);                //Does not compile!
List<Point2D> line = new List<Point2D>(1000000);
for (int i = 0; i < 1000000; ++i) {
  line.Add(new Point2D(i, i));    //No boxing, no references stored
}
```

Almost magically, the generic collection is type safe (in that it does not allow inappropriate elements to be stored in it) and requires no boxing for value types. Even the internal storage — the items array — adapts itself accordingly to the generic type argument: when `T` is `Point2D`, the `items` array is a `Point2D[]`, which stores *values* and not references. We will unfold some of this magic later, but for now we have an effective language-level solution for the basic problems of generic programming.

This solution alone appears to be insufficient when we require some additional capabilities from the generic parameters. Consider a method that performs a binary search in a sorted array. A completely generic version of it would not be capable of doing the search, because `System.Object` is not equipped with any comparison-related facilities:

```
public static int BinarySearch<T>(T[] array, T element) {
  //At some point in the algorithm, we need to compare:
  if (array[x] < array[y]) {
   ...
  }
}
```

`System.Object` does not have a static `operator <` , which is why this method simply fails to compile! Indeed, we have to prove to the compiler that for every generic type argument we provide to the method, it will be able to resolve all method calls (including operators) on it. This is where generic constraints enter the picture.

Generic Constraints

Generic constraints indicate to the compiler that only *some* types are allowed as generic type arguments when using a generic type. There are five types of constraints:

```
//T must implement an interface:
public int Format(T instance) where T : IFormattable {
  return instance.ToString("N", CultureInfo.CurrentUICulture);
  //OK, T must have IFormattable.ToString(string, IFormatProvider)
}
//T must derive from a base class:
public void Display<T>(T widget) where T : Widget {
  widget.Display(0, 0);
  //OK, T derives from Widget which has the Display(int, int) method
}
//T must have a parameterless constructor:
public T Create<T>() where T : new() {
  return new T();
  //OK, T has a parameterless constructor
  //The C# compiler compiles 'new T()' to a call to Activator.CreateInstance<T>(),
  //which is sub-optimal, but there is no IL equivalent for this constraint
}
//T must be a reference type:
public void ReferencesOnly<T>(T reference) where T : class

//T must be a value type:
public void ValuesOnly<T>(T value) where T : struct
```

If we revisit the binary search example, the interface constraint proves to be very useful (and indeed, it is the most often used kind of constraint). Specifically, we can require `T` to implement `IComparable`, and compare array elements using the `IComparable.CompareTo` method. However, `IComparable` is not a generic interface, and its `CompareTo` method accepts a `System.Object` parameter, incurring a boxing cost for value types. It stands to reason that there should be a generic version of `IComparable`, `IComparable<T>`, which serves here perfectly:

```
//From the .NET Framework:
public interface IComparable<T> {
  int CompareTo(T other);
}
public static int BinarySearch<T>(T[] array, T element) where T : IComparable<T> {
  //At some point in the algorithm, we need to compare:
  if (array[x].CompareTo(array[y]) < 0) {
   ...
  }
}
```

This binary search version does not incur boxing when comparing value type instances, works with any type that implements `IComparable<T>` (including all the built-in primitive types, strings, and many others), and is completely type safe in that it does not require runtime discovery of comparison capabilities.

INTERFACE CONSTRAINTS AND IEQUATABLE<T>

In Chapter 3 we have seen that a critical performance optimization for value types is to override the `Equals` method and implement the `IEquatable<T>` interface. Why is this interface so important? Consider the following code:

```
public static void CallEquals<T>(T instance) {

      instance.Equals(instance);
   }
```

The `Equals` call inside the method defers to a virtual call to `Object.Equals`, which accepts a `System.Object` parameter and causes boxing on value types. This is the only alternative that the C# compiler considers guaranteed to be present on every `T` we use. If we want to convince the compiler that `T` has an `Equals` method that accepts `T`, we need to use an explicit constraint:

```
//From the .NET Framework:
public interface IEquatable<T> {
   bool Equals(T other);
}
public static void CallEquals<T>(T instance) where T : IEquatable<T> {
   instance.Equals(instance);
}
```

Finally, you might want to allow callers to use any type as `T`, but defer to the `IEquatable<T>` implementation if `T` provides it, because it does not require boxing and is more efficient. What `List<T>` does in this regard is fairly interesting. If `List<T>` required an `IEquatable<T>` constraint on its generic type parameter, it wouldn't be useful for types that don't implement that interface. Therefore, `List<T>` does not have an `IEquatable<T>` constraint. To implement the `Contains` method (and other methods that require comparing objects for equality), `List<T>` relies on an *equality comparer*—a concrete implementation of the abstract `EqualityComparer<T>` class (which, incidentally, implements the `IEqualityComparer<T>` interface, used directly by some collections, including `HashSet<T>` and `Dictionary<K,V>`).

When `List<T>.Contains` needs to invoke `Equals` on two elements of the collection, it uses the `EqualityComparer<T>.Default` static property to retrieve an equality comparer implementation suitable for comparing instances of `T`, and calls its `Equals(T, T)` virtual method. It's the `EqualityComparer<T>.CreateComparer` private static method that creates the appropriate equality comparer the first time it's requested and subsequently caches it in a static field. When `CreateComparer` sees that `T` implements `IEquatable<T>`, it returns an instance of `GenericEqualityComparer<T>`, which has an `IEquatable<T>` constraint and invokes `Equals` through the interface. Otherwise, `CreateComparer` resorts to the `ObjectEqualityComparer<T>` class, which has no constraints on `T` and invokes the `Equals` virtual method provided by `Object`.

This trick employed by `List<T>` for equality checking can be useful for other purposes as well. When a constraint is available, your generic class or method can use a potentially more efficient implementation that does not resort to runtime type checking.

Tip As you see, there are no generic constraints to express mathematical operators, such as addition or multiplication. This means that you can't write a generic method that uses an expression such as `a+b` on generic parameters. The standard solution for writing generic numeric algorithms is to use a helper struct that implements an `IMath<T>` interface with the required arithmetic operations, and instantiate this struct in the generic method. For more details, see Rüdiger Klaehn's CodeProject article, "Using generics for calculations," available at http://www.codeproject.com/Articles/8531/Using-generics-for-calculations).

Having examined most syntactic features of generics in C#, we turn to their runtime implementation. Before we can concern ourselves with this matter, it's paramount to ask whether there *is* a runtime representation of generics — as we will see shortly, C++ templates, a similar mechanism, have no runtime representation to speak of. This question is easily answered if you look at the wonders Reflection can do with generic types *at runtime*:

```
Type openList = typeof(List<>);
Type listOfInt = openList.MakeGenericType(typeof(int));
IEnumerable<int> ints = (IEnumerable<int>)Activator.CreateInstance(listOfInt);

Dictionary<string, int> frequencies = new Dictionary<string, int>();
Type openDictionary = frequencies.GetType().GetGenericTypeDefinition();
Type dictStringToDouble = openDictionary.MakeGenericType(typeof(string), typeof(double));
```

As you see, we can dynamically create generic types from an existing generic type and parameterize an "open" generic type to create an instance of a "closed" generic type. This demonstrates that generics are first-class citizens and have a runtime representation, which we now survey.

Implementation of CLR Generics

The syntactic features of CLR generics are fairly similar to Java generics and even slightly resemble C++ templates. However, it turns out that their internal implementation and the limitations imposed upon programs using them are very different from Java and C++. To understand these differences we should briefly review Java generics and C++ templates.

Java Generics

A generic class in Java can have generic type parameters, and there is even a constraint mechanism quite similar to what .NET has to offer (bounded type parameters and wildcards). For example, here's a first attempt to convert our `List<T>` to Java:

```
public class List<E> {
  private E[] items;
  private int size;
  public List(int initialCapacity) {
   items = new E[initialCapacity];
  }
  public void Add(E item) {
   if (size < items.Length – 1) {
   items[size] = item;
   ++size;
   } else {
   //Allocate a larger array, copy the elements, then place 'item' in it
   }
  }
  public E getAt(int index) {
   if (index < 0 || index >= size) throw IndexOutOfBoundsException(index);
   return items[index];
  }
  //Many more methods omitted for brevity
}
```

Unfortunately, this code does not compile. Specifically, the expression `new E[initialCapacity]` is not legal in Java. The reason has to do with the way Java compiles generic code. The Java compiler removes any mentions of the generic type parameter and replaces them with `java.lang.Object`, a process called *type erasure*. As a result, there is only one type at runtime — `List`, the *raw* type — and any information about the generic type argument provided is lost. (To be fair, by using type erasure Java retains binary compatibility with libraries and applications that were created before generics — something that .NET 2.0 does not offer for .NET 1.1 code.)

Not all is lost, though. By using an `Object` array instead, we can reconcile the compiler and still have a type-safe generic class that works well at compilation time:

```
public class List<E> {
  private Object[] items;
  private int size;
  public void List(int initialCapacity) {
   items = new Object[initialCapacity];
  }
  //The rest of the code is unmodified
}
List<Employee> employees = new List<Employee>(7);
employees.Add(new Employee("Kate"));
employees.Add(42); //Does not compile!
```

However, adopting this approach in the CLR voices a concern: what is to become of value types? One of the two reasons for introducing generics was that we wanted to avoid boxing at any cost. Inserting a value type to an array of objects requires boxing and is not acceptable.

C++ Templates

Compared to Java generics, C++ templates may appear enticing. (They are also extremely powerful: you may have heard that the template resolution mechanism in itself is Turing-complete.) The C++ compiler does not perform type erasure — quite the opposite — and there's no need for constraints, because the compiler is happy to compile whatever you throw in its way. Let's start with the list example, and then consider what happens with constraints:

```
template <typename T>
class list {
private:
  T* items;
  int size;
  int capacity;
public:
  list(int initialCapacity) : size(0), capacity(initialCapacity) {
   items = new T[initialCapacity];
  }
  void add(const T& item) {
   if (size < capacity) {
   items[size] = item;
   ++size;
   } else {
   //Allocate a larger array, copy the elements, then place 'item' in it
   }
  }
  const T& operator[](int index) const {
   if (index < 0 || index >= size) throw exception("Index out of bounds");
   return items[index];
  }
  //Many more methods omitted for brevity
};
```

The list template class is completely type-safe: *every* instantiation of the template creates a new class that uses the template definition as a… template. Although this happens under the hood, here is an example of what it *could* look like:

```
//Original C++ code:
list<int> listOfInts(14);
//Expanded by the compiler to:
class __list__int {
private:
  int* items;
  int size;
  int capacity;
public:
  __list__int(int initialCapacity) : size(0), capacity(initialCapacity) {
   items = new int[initialCapacity];
  }
};
__list__int listOfInts(14);
```

Note that the `add` and `operator[]` methods were not expanded — the calling code did not use them, and the compiler generates only the parts of the template definition that are used for the particular instantiation. Also note that the compiler does not generate *anything* from the template definition; it waits for a specific instantiation before it produces any code.

This is why there is no need for constraints in C++ templates. Returning to our binary search example, the following is a perfectly reasonable implementation:

```
template <typename T>
int BinarySearch(T* array, int size, const T& element) {
  //At some point in the algorithm, we need to compare:
  if (array[x] < array[y]) {
   ...
  }
}
```

There's no need to prove anything to the C++ compiler. After all, the template definition is meaningless; the compiler waits carefully for any instantiations:

```
int numbers[10];
BinarySearch(numbers, 10, 42); //Compiles, int has an operator <
class empty {};
empty empties[10];
BinarySearch(empties, 10, empty()); //Does not compile, empty does not have an operator <
```

Although this design is extremely tempting, C++ templates have unattractive costs and limitations, which are undesirable for CLR generics:

- Because the template expansion occurs at compile-time, there is no way to share template instantiations between different binaries. For example, two DLLs loaded into the same process may have separate compiled versions of `list<int>`. This consumes a large amount of memory and causes long compilation times, by which C++ is renowned.
- For the same reason, template instantiations in two different binaries are considered incompatible. There is no clean and supported mechanism for exporting template instantiations from DLLs (such as an exported function that returns `list<int>`).
- There is no way to produce a binary library that contains template definitions. Template definitions exist only in source form, as header files that can be `#include`d into a C++ file.

Generics Internals

After giving ample consideration to the design of Java generics C++ templates, we can understand better the implementation choice for CLR generics. CLR generics are implemented as follows. Generic types — even open ones, like `List<>` — are first-class runtime citizens. There is a method table and an EEClass (see Chapter 3) for each generic type and a `System.Type` instance can be produced as well. Generic types can be exported from assemblies and only a single definition of the generic type exists at compile-time. Generic types are not expanded at compile-time, but as we have seen the compiler makes sure that any operation attempted on generic type parameter instances is compatible with the specified generic constraints.

When the CLR needs to create an instance of a closed generic type, such as `List<int>`, it creates a method table and EEClass based on the open type. As always, the method table contains method pointers, which are compiled on the fly by the JIT compiler. However, there is a crucial optimization here: compiled method bodies on closed generic types that have reference type parameters can be shared. To digest this, let's consider the `List<T>.Add` method and try to compile it to x86 instructions when `T` is a reference type:

```
//C# code:
public void Add(T item) {
  if (size < items.Length – 1) {
   items[size] = item;
   ++size;
  } else {
   AllocateAndAddSlow(item);
  }
}
; x86 assembly when T is a reference type
; Assuming that ECX contains ‘this’ and EDX contains ‘item’, prologue and epilogue omitted
mov eax, dword ptr [ecx+4] ; items
mov eax, dword ptr [eax+4] ; items.Length
dec eax
cmp dword ptr [ecx+8], eax ; size < items.Length - 1
jge AllocateAndAddSlow
mov eax, dword ptr [ecx+4]
mov ebx, dword ptr [ecx+8]
mov dword ptr [eax+4*ebx+4], edx ; items[size] = item
inc dword ptr [eax+8] ; ++size
```

It's clear that the method's code does not depend on `T` in any way, and will work exactly as well for any reference type. This observation allows the JIT compiler to conserve resources (time and memory) and share the method table pointers for `List<T>.Add` in all method tables where `T` is a reference type.

Note This idea requires some further refinement, which we will not carry out. For example, if the method's body contained a `new T[10]` expression, it would require a separate code path for each `T`, or at least a way of obtaining `T` at runtime (e.g., through an additional hidden parameter passed to the method). Additionally, we haven't considered how constraints affect the code generation — but you should be convinced by now that invoking interface methods or virtual methods through base classes requires the same code regardless of the type.

The same idea does not work for value types. For example, when `T` is `long`, the assignment statement `items[size] = item` requires a different instruction, because 8 bytes must be copied instead of 4. Even larger value types may even require more than one instruction; and so on.

To demonstrate Figure 5-2 in a simple setting, we can inspect using SOS the method tables of closed generic types that are all realizations of the same open generic type. For example, consider a `BasicStack<T>` class with only `Push` and `Pop` methods, as follows:

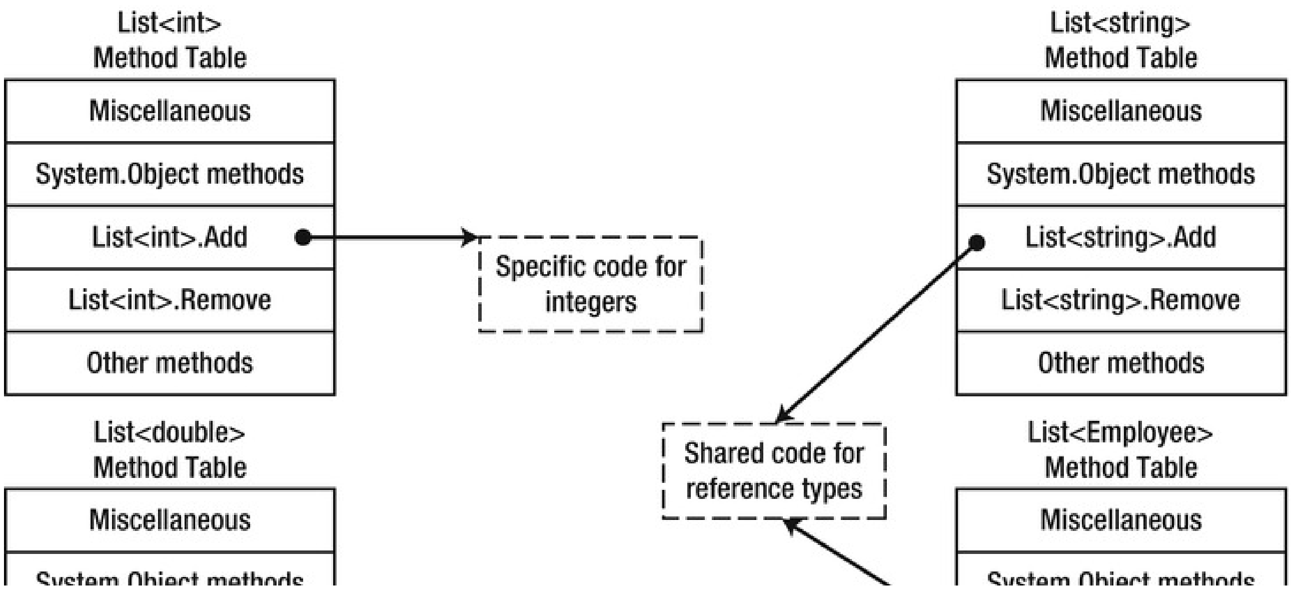

Figure 5-2 . The Add method table entries for reference type realizations of List<T> have shared pointers to a single method implementation, whereas the entries for value type realizations have separate versions of the code

```
class BasicStack<T> {
   private T[] items;
   private int topIndex;
   public BasicStack(int capacity = 42) {
   items = new T[capacity];
   }
   public void Push(T item) {
   items[topIndex++] = item;
   }
   public T Pop() {
   return items[--topIndex];
  }
}
```

The method tables for `BasicStack<string>`, `BasicStack<int[]>`, `BasicStack<int>`, and `BasicStack<double>` are listed below. Note that the method table entries (i.e. code addresses) for the closed types with a reference type generic type argument are shared, and for the value types are not:

```
0:004> !dumpheap -stat
...
00173b40 1 16 BasicStack`1[[System.Double, mscorlib]]
```

```
00173a98 1 16 BasicStack`1[[System.Int32, mscorlib]]
00173a04 1 16 BasicStack`1[[System.Int32[], mscorlib]]
001739b0 1 16 BasicStack`1[[System.String, mscorlib]]
...
0:004> !dumpmt -md 001739b0
EEClass: 001714e0
Module: 00172e7c
Name: BasicStack`1[[System.String, mscorlib]]
...
MethodDesc Table
   Entry MethodDe JIT Name
...
00260360 00173924 JIT BasicStack`1[[System.__Canon, mscorlib]].Push(System.__Canon)
00260390 0017392c JIT BasicStack`1[[System.__Canon, mscorlib]].Pop()
0:004> !dumpmt -md 00173a04
EEClass: 001714e0
Module: 00172e7c
Name: BasicStack`1[[System.Int32[], mscorlib]]
...
MethodDesc Table
   Entry MethodDe JIT Name
...
00260360 00173924 JIT BasicStack`1[[System.__Canon, mscorlib]].Push(System.__Canon)
00260390 0017392c JIT BasicStack`1[[System.__Canon, mscorlib]].Pop()
0:004> !dumpmt -md 00173a98
EEClass: 0017158c
Module: 00172e7c
Name: BasicStack`1[[System.Int32, mscorlib]]
...
MethodDesc Table
   Entry MethodDe JIT Name
...
002603c0 00173a7c JIT BasicStack`1[[System.Int32, mscorlib]].Push(Int32)
002603f0 00173a84 JIT BasicStack`1[[System.Int32, mscorlib]].Pop()
0:004> !dumpmt -md 00173b40
EEClass: 001715ec
Module: 00172e7c
Name: BasicStack`1[[System.Double, mscorlib]]
...
MethodDesc Table
   Entry MethodDe JIT Name
...
00260420 00173b24 JIT BasicStack`1[[System.Double, mscorlib]].Push(Double)
00260458 00173b2c JIT BasicStack`1[[System.Double, mscorlib]].Pop()
```

Finally, if we inspect the actual method bodies, it becomes evident that the reference type versions do not depend at all on the actual type (all they do is move references around), and the value type versions *do* depend on the type. Copying an integer, after all, is different from copying a double floating-point number. Below are the disassembled versions of the `Push` method, with the line that actually moves the data around highlighted:

```
0:004> !u 00260360
Normal JIT generated code
BasicStack`1[[System.__Canon, mscorlib]].Push(System.__Canon)
00260360 57              push    edi
00260361 56              push    push    esi
00260362 8b7104          push    mov     esi,dword ptr [ecx+4]
00260365 8b7908          push    mov     edi,dword ptr [ecx+8]
00260368 8d4701          push    lea     eax,[edi+1]
0026036b 894108          push    mov     dword ptr [ecx+8],eax
0026036e 52              push    edx
```

```
0026036f 8bce              push     mov    ecx,esi
00260371 8bd7              push     mov    edx,edi
00260373 e8f4cb3870        push    call    clr!JIT_Stelem_Ref (705ecf6c)
00260378 5e                push    pop     esi
00260379 5f                push    pop     edi
0026037a c3                push    ret

0:004> !u 002603c0
Normal JIT generated code
BasicStack`1[[System.Int32, mscorlib]].Push(Int32)
002603c0 57          push   edi
002603c1 56          push   esi
002603c2 8b7104      mov    esi,dword ptr [ecx+4]
002603c5 8b7908      mov    edi,dword ptr [ecx+8]
002603c8 8d4701      lea    eax,[edi+1]
002603cb 894108      mov    dword ptr [ecx+8],eax
002603ce 3b7e04      cmp    edi,dword ptr [esi+4]
002603d1 7307        jae    002603da
002603d3 8954be08    mov    dword ptr [esi+edi*4+8],edx
002603d7 5e          pop    esi
002603d8 5f          pop    edi
002603d9 c3          ret
002603da e877446170  call   clr!JIT_RngChkFail (70874856)
002603df cc    int    3

0:004> !u 00260420
Normal JIT generated code
BasicStack`1[[System.Double, mscorlib]].Push(Double)
00260420 56            push   esi
00260421 8b5104        mov    edx,dword ptr [ecx+4]
00260424 8b7108        mov    esi,dword ptr [ecx+8]
00260427 8d4601        lea    eax,[esi+1]
0026042a 894108        mov    dword ptr [ecx+8],eax
0026042d 3b7204        cmpyg  esi,dword ptr [edx+4]
00260430 730c          jae    0026043e
00260432 dd442408      fld    qword ptr [esp+8]
00260436 dd5cf208     fstp    qword ptr [edx+esi*8+8]
0026043a 5e           pop     esi
0026043b c20800       ret     8
0026043e e813446170   call    clr!JIT_RngChkFail (70874856)
00260443 cc    int    3
```

We have already seen that the .NET generics implementation is fully type-safe at compile-time. It only remains to ascertain that there is no boxing cost incurred when using value types with generic collections. Indeed, because the JIT compiler compiles a separate method body for each closed generic type where the generic type arguments are value types, there is no need for boxing.

To summarize, .NET generics offer significant advantages when contrasted with Java generics or C++ templates. The generic constraints mechanism is somewhat limited compared to the Wild West that C++ affords, but the flexibility gains from sharing generic types across assemblies and the performance benefits from generating code on demand (and sharing it) are overwhelming.