2

Valuing Type Safety

This book is filled with references to the importance of type safety. The term gets thrown around very loosely inside and outside the world of generics. So much so, that it seems like its meaning is often lost in the shuffle as a core value for many developers. Now, with generics, it's worth reexamining the value of type safety because it's one of the motivating factors that influenced the introduction of this new language feature. This chapter revisits the origins of type safety and discusses some of the unsafe trends that have become a common occurrence. Certainly, this is an area where there may be some disagreement. However, it's an area that needs to be discussed as part of sharpening your awareness and understanding the impact generics will have on your everyday approach to designing and building solutions.

Motivation

Types have to matter. With every class you write, you need to be focused on how that class represents itself to clients. Each time clients touch the interfaces of your class, they are binding to the specific types exposed in the signature of that interface. As such, you need to be concerned about the type-safety implications that accompany each of these interactions. Does your interface provide a clear set of types that make every attempt to eliminate ambiguity, or does your interface favor generality at the expense of type safety?

From my perspective, a great deal of what generics has to offer is focused squarely on allowing you to achieve a much greater level of type safety without having to compromise on generality (or bloat your code with more specialized classes). Generics should, in some respects, force you to apply a higher standard to the classes you write and consume. They should put you in a position where you look at the type safety of each interface with a significantly higher level of scrutiny than you might have in the pre-generics era.

As best I can tell, this fundamental mindset is sometimes lost in the discussions surrounding generics. Whenever developers look at a new language feature, they often ask, "What new functionality can I build with this feature that I couldn't build before?" Though generics do enable *new*

capabilities, that's not the point. Generics aren't just about doing something new — they're about doing something *better*. Through generics, you should be able to bring a new dimension of type safety and expressiveness to your code that will undoubtedly improve its quality, usability, and maintainability.

The goal, then, as you move through this chapter, is to bring some light to how generics can influence the type safety of your code. There are simply too many permutations of type-safety scenarios to address them all. That's not my approach. I just want to provide enough insight to establish a theme that I hope influences how you look at applying generics to your new and existing solutions.

The truth is, though, if you don't see code and value its ability to adequately convey and constrain its types, you are likely to miss out on one of the key benefits of generics.

Least Common Denominator Programming

In the early days of Java, I remember discussing templates with a few of the C++ converts. Whenever the conversation turned to templates (the C++ variation of generics), they usually said: "I don't need templates because everything in Java descends from an object." And, I'm assuming this same train of thought has actually carried forward into some segment of the .NET community, where every class is also rooted in a common object type.

This general mindset has always puzzled me. I understand that having everything rooted in a single object hierarchy enables some generality. It even makes sense to me that a number of classes would leverage this reality. At the same time, I don't think it would be accurate to view this feature as somehow replacing or offsetting the need for generics.

The `Object` data type, in fact, can end up being quite a crutch. Developers will leverage it in a number of situations where they want to provide a highly generalized interface that accepts any number of different data types. `ArrayList` is the great example of a class that takes this approach. As a data container, it needs to be able to hold any type of object. So, it's forced to use the `Object` data type to represent the types it holds. You'll also see situations where developers will accept or return `Object` parameters in an interface that needs to handle a wide variety of unrelated objects.

This use of the `Object` type is natural and expected. If `ArrayList` and other classes didn't leverage this mechanism, they would be forced to introduce class after class of specialized types to support each unique type they needed to manage. I wouldn't want to see `DoubleArrayList`, `StringArrayList`, and so on. That would often be too high of a price to pay for type safety.

So, as you code, you constantly face this question of deciding when it might be appropriate to leverage the `Object` data type and, each time you make the compromise, you also compromise the type safety of your code. With generics, the idea is to break this pattern of least common denominator coding. For example, the BCL now replaces those non-generic, type-safety-hating classes from the `System.Collections` namespace with new, type-safe versions in the `System.Collections.Generic` namespace.

In many respects, I see a generic type, `T`, as the direct replacement for an `Object` data type. By using `T`, you are still indicating that any type (value or reference) can be accepted, which allows you to retain the generality you needed. At the same time, unlike the `Object`, `T` will represent a binding to a very specific data type. So, you get the best of both worlds.

A Basic Example

Type safety is likely a term you happened upon quite frequently in your travels as an object-oriented programmer. And, for you, the concept may already be crystallized. That said, I want to be sure we're on equal footing before examining some of the broader issues surrounding type safety and generics. So, to establish some common ground, let's look at a simple scenario that provides a very basic example of the importance of type safety.

The example you'll construct here consists of an object hierarchy with a `Person` class at the root and two descendant classes, `Customer` and `Employee`. The `Person` class provides an abstraction of those attributes that are common to every person. In this example, these shared attributes are represented by the `Id`, `Name`, and `Status` properties of the `Person` class.

The `Customer` and `Employee` classes also add their own specializations and behavior. Specifically, each of these classes also has a one-to-many relationship with another class. A `Customer` is associated with one or more `Order`s and an `Employee` contains references to one or more "child" `Employee` objects that represent those employees that are managed by a specific person.

Now, in working with these `Customer` and `Employee` objects, assume you've identified several places in your code that are providing general-purpose handling of `Person` objects. To further promote this generality, you've decided you also would like to allow clients of your `Person` class to access the items associated with a `Person`. To accommodate this, you've moved one more property, `Items`, up into your `Person` class.

Unfortunately, because the classes associated with each `Person` don't necessarily share a common base class, you are forced to represent this new property as an `Object` type. The beauty of this approach is that your `Person` class now exposes a generalized approach to exposing an interface all clients can use to retrieve the items associated with any type of `Person`.

Here's how this `Person` object might be represented:

```
[VB Code]
Public Class Person
    Public Const ACTIVE_STATUS As Int32 = 1
    Public Const INACTIVE_STATUS As Int32 = 2
    Public Const NEW_STATUS As Int32 = 3

    Private _name As String
    Private _id As String
    Private _status As Int32
    Private _items As ArrayList

    Public Sub New(ByVal Id As String, ByVal Name As String, ByVal Status As Int32)
        Me._id = Id
        Me._name = Name
        Me._status = Status
        Me._items = New ArrayList()
    End Sub

    Public ReadOnly Property Id() As String
        Get
```

```
            Return Me._id
        End Get
    End Property

    Public ReadOnly Property Name() As String
        Get
            Return Me._name
        End Get
    End Property

    Public ReadOnly Property Status() As Int32
        Get
            Return Me._status
        End Get
    End Property

    Public ReadOnly Property Items() As Object()
        Get
            Return Me._items.ToArray()
        End Get
    End Property

    Public Sub AddItem(ByVal newItem As Object)
        Me._items.Add(newItem)
    End Sub
End Class
```

```
[C# code]
public class Person {
    public const int ACTIVE_STATUS = 1;
    public const int INACTIVE_STATUS = 2;
    public const int NEW_STATUS = 3;

    private string _id;
    private string _name;
    private int _status;
    private ArrayList _items;

    public Person(String Id, String Name, int Status) {
        this._id = Id;
        this._name = Name;
        this._status = Status;
        this._items = new ArrayList();
    }

    public string Id {
        get { return this._id; }
    }

    public string Name {
        get { return this._name; }
    }
```

```
    public int Status {
        get { return this._status; }
    }

    public Object[] Items {
        get { return this._items.ToArray(); }
    }

    public void AddItem(Object newItem) {
        this._items.Add(newItem);
    }
}
```

On the surface, there's nothing glaringly wrong with this class. Its interface is intuitive enough. It does have some type-safety issues, though. Some are obvious and some not.

The `Status` property represents the simplest form of type-safety violation and likely falls into the "obvious" bucket. Even though constants are used to define the valid range of values that can be assigned to this property, you cannot prevent clients from setting it to *any* valid integer value. By making this property an integer, you've really limited your ability to enforce any kind of compile- or run-time type checking of this value. I guess you could actually validate it against the known range at run-time, but that's awkward at best. The real type-safe solution here would be to make your property an `Enum`.

So, the `Status` provides a simple, clean example of why type safety is important. However, that scenario didn't require generics to be resolved. To see where generics would be applied, you must first assemble some sample code that exercises the `Person` object. Let's start with some simple code that creates a `Customer` object and populates it with some orders (the code for the `Customer` object is not shown here, but it is available as part of the complete examples that can be downloaded from the Wrox Web site).

```
[VB code]
Public Function PopulateCustomerCollection() As ArrayList
    Dim custColl As New ArrayList()
    Dim cust As New Customer("1", "Ron Livingston", 1)
    cust.AddItem(New Order(DateTime.Parse("10/1/2004"), "SL", "Swingline Stapler"))
    cust.AddItem(New Order(DateTime.Parse("10/03/2004"), "XR", "Xerox Copier"))
    cust.AddItem(New Order(DateTime.Parse("10/07/2004"), "FX", "Fax Paper"))
    custColl.Add(cust)

    cust = New Customer("2", "Milton Waddams", 2)
    cust.AddItem(New Order(DateTime.Parse("11/04/2004"), "PR-061", "Printer"))
    cust.AddItem(New Order(DateTime.Parse("11/07/2004"), "3H-24", "3-hole punch"))
    cust.AddItem(New Order(DateTime.Parse("12/12/2004"), "DSK-36", "CDRW Disks"))
    custColl.Add(cust)

    cust = New Customer("3", "Bill Lumberg", 3)
    cust.AddItem(New Order(DateTime.Parse("10/01/2004"), "WST4", "Waste basket"))
    custColl.Add(cust)

    Return custColl
End Function
```

```
[C# code]
public ArrayList PopulateCustomerCollection() {
    ArrayList custColl = new ArrayList();
    Customer cust = new Customer("1", "Ron Livingston", 1);
    cust.AddItem(new Order(DateTime.Parse("10/1/2004"),"SL", "Swingline Stapler"));
    cust.AddItem(new Order(DateTime.Parse("10/03/2004"), "XR", "Xerox Copier"));
    cust.AddItem(new Order(DateTime.Parse("10/07/2004"), "FX", "Fax Paper"));
    custColl.Add(cust);

    cust = new Customer("2", "Milton Waddams", 2);
    cust.AddItem(new Order(DateTime.Parse("11/04/2004"), "PR-061", "Printer"));
    cust.AddItem(new Order(DateTime.Parse("11/07/2004"), "3H-24", "3-hole punch"));
    cust.AddItem(new Order(DateTime.Parse("12/12/2004"), "DSK-36", "CD-RW Disks"));
    custColl.Add(cust);

    cust = new Customer("3", "Bill Lumberg", 3);
    cust.AddItem(new Order(DateTime.Parse("10/01/2004"), "WST4", "Waste basket" ));
    custColl.Add(cust);

    return custColl;
}
```

Now, you're thinking, what's wrong with this? The answer is: nothing. This code is perfectly fine as it is. However, think about what the interface of the `Person` object enables here. Imagine if you were to change this same code to the following:

```
[VB code]
Dim cust As New Customer("1", "Ron Livingston", 1)
cust.AddItem(New Dog("Sparky", "Mutt"))
```

```
[C# code]
Customer cust = new Customer("1", "Ron Livingston", 1);
cust.AddItem(new Dog("Sparky", "Mutt"));
```

Instead of associating orders with your `Customer`, you've now associated a set of `Dog` objects with your `Customer`. Because the interface of your class must accept `Object`s as its incoming type, there's nothing that prevents you from adding *any* flavor of object to your `Customer` — even if the implied rules of your `Customer` indicate this is invalid. It's not until someone starts to consume your `Customer` object that they will catch the error that this introduces.

In fact, let's look at some of the type-safety issues that this class represents from the consumer's perspective. The following example creates a method that iterates over a list of customers, dumping out the information for each customer and their associated orders:

```
 [VB code]
Public Sub DisplayCustomers(ByVal customers As ArrayList)
    For custIdx As Int32 = 0 To (customers.Count - 1)
        Dim cust As Customer = customers(custIdx)
        Console.Out.WriteLine("Customer-> ID: {0}, Name: {1}", cust.Id, cust.Name)
        Dim orders() As Object = DirectCast(cust.Items, Object())

        For orderIdx As Int32 = 0 To (orders.Length - 1)
            Dim ord As Order = DirectCast(orders(orderIdx), Order)
```

```
            Console.Out.WriteLine("  Order-> Date: {0}, Item: {1}, Desc: {2}", _
                                    ord.OrderDate, ord.ItemId, ord.Description)
        Next
    Next
End Sub
```

```
[C# code]
public void DisplayCustomers(ArrayList customers) {
    for (int custIdx = 0; custIdx < customers.Count; custIdx++) {
        Customer cust = (Customer)customers[custIdx];
        Console.Out.WriteLine("Customer-> ID: {0}, Name: {1}", cust.Id, cust.Name);

        Object[] orders = (Object[])cust.Items;
        for (int orderIdx = 0; orderIdx < orders.Length; orderIdx++) {
            Order ord = (Order)orders[orderIdx];
            Console.Out.WriteLine("  Order-> Date: {0}, Item: {1}, Desc: {2}",
                                    ord.OrderDate, ord.ItemId, ord.Description);
        }
    }
}
```

Now, in looking at these methods, you can see where moving your items collection up into the `Person` class is causing some real type-safety concerns. Its declaration starts everything off on the wrong foot. It uses an `ArrayList` to hold the incoming list of customers and, because `ArrayList`s can only represent their contents as objects, you're required to cast each object to a `Customer` as it comes out of the list. So, if a client happens to pass in an `ArrayList` of employees, your method will accept it and then toss an exception when you attempt to cast one of its items to a `Customer`. Strike one.

The other area of concern is centered on the processing of the orders associated with each `Customer`. You'll notice here that, as you get the array of orders from the `Items` property of the `Customer`, you are required to cast the returned array to an array of `Object`s. That's right—because the `Items` property returns an array of `Object`s, you cannot directly cast this to an array of orders, which is what you really want. Strike two.

Finally, because you're dealing with an array of `Object`s here, you're forced to cast each object to an `Order` as it is extracted from this array. Strike three.

As you look at this line of thought, I imagine you might have a few reactions. First, you might take the position that this is just the cost of generality and that, as long as you're careful, a few casts here and there aren't exactly dangerous. Still, it seems to defeat the purpose of representing your `customer`s and `order`s with these fairly expressive interfaces, only to push the value and safety that comes with this aside to achieve some higher level of generality. The introduction of these casts also creates yet one more area for producing errors and maintenance overhead.

The other angle here might be to suggest that you could avoid a great deal of this casting by adding specific interfaces in your `Customer` and `Employee` classes that returned the appropriate types. This would allow you to keep the generality in your base class and would simply cast the items to their specific types on the way out to a client. This is a reasonable compromise and is likely how many people have historically addressed a problem of this nature. It certainly limits each client's exposure to the `Object` representation of the `Items` property. Still, using `Object`s to represent these items is troubling from a pure type-safety perspective.

Applying Generics

The question that remains is, how can generics be applied to overcome some of the type-safety problems illustrated in this example? You still want your `Person` class to expose an interface for retrieving each of its items, but you want the types of those items to be safe. Because generics give you a way to parameterize your types, you can use them, in this scenario, to parameterize your `Person` class, allowing it to accept a type parameter that will specify the type of the elements collected by the `Items` property. The resulting, generically improved `Person` class now appears as follows:

```
    [VB code]
Public Class Person(Of T)
    Public Enum StatusType
        Active = 1
        Inactive = 2
        IsNew = 3
    End Enum

    Private _name As String
    Private _id As String
    Private _status As StatusType
    Private _items As List(Of T)

    Public Sub New(ByVal Id As String, ByVal Name As String, _
                                          ByVal Status As StatusType)
        Me._id = Id
        Me._name = Name
        Me._status = Status
        Me._items = New List(Of T)
    End Sub

    Public ReadOnly Property Id() As String
        Get
            Return Me._id
        End Get
    End Property

    Public ReadOnly Property Name() As String
        Get
            Return Me._name
        End Get
    End Property

    Public ReadOnly Property Status() As StatusType
        Get
            Return Me._status
        End Get
    End Property

    Public ReadOnly Property Items() As T()
        Get
            Return Me._items.ToArray()
        End Get
    End Property
```

```
        Public Sub AddItem(ByVal newItem As T)
            Me._items.Add(newItem)
        End Sub
    End Class

    [C# code]
    public class Person<T> {
        public enum StatusType {
            Active = 1,
            Inactive = 2,
            IsNew = 3
        };

        private string _id;
        private string _name;
        private StatusType _status;
        private List<T> _items;

        public Person(String Id, String Name, StatusType Status) {
            this._id = Id;
            this._name = Name;
            this._status = Status;
            this._items = new List<T>();
        }

        public string Id {
            get { return this._id; }
        }

        public string Name {
            get { return this._name; }
        }

        public StatusType Status {
            get { return this._status; }
        }

        public T[] Items {
            get { return this._items.ToArray(); }
        }

        public void AddItem(T newItem) {
            this._items.Add(newItem);
        }
    }
```

That's only step one in the purification of this class. You also need to change the internal representation of the `items` data member. Instead of clinging to that old, type-ignorant `ArrayList`, you can use one of the new generic `List` collections (from the `System.Collections.Generic` namespace described in Chapter 8, "BCL Generics") to bring a greater level of type safety to this data member. To be complete, the `Status` property is also changed from an integer to an `enum` type.

Finally, to round out this transformation, you'll notice that the parameterization of the `Person` class allows you to change the `AddItem()` method to enforce type checking. Now, each object type that gets

added must match the type of the type parameter, `T`, to be considered valid. No more adding dogs to customers.

An added bonus associated with this approach is that clients are still not required to have any awareness of the fact that you've applied generics to solve this problem. The `Customer` and `Employee` classes, which descend from `Person`, simply specify the type of their related items as part of their inheritance declarations. Here's a snippet of these class declarations to clarify this point:

```
[VB code]
Public Class Customer
    Inherits Person(Of Order)
    ...
End Class

Public Class Employee
    Inherits Person(Of Employee)
    ...
End Class
```

```
[C# code]
public class Customer : Person<Order> {
    ...
}

public class Employee : Person<Employee> {
    ...
}
```

As you can see, even though you've leveraged generics to add type safety to your `Person` class, these two classes retain the same interface they supported under the non-generic version. In fact, the client code used to populate the `Customer` and `Employee` structures would not require any modifications (with the exception of the change that was introduced to make `Status` an `enum`).

Although the code to populate the `Customer` and `Employee` classes was unscathed as a result of making `Person` generic, the code that was used earlier to dump information about customers does require changes (all of them for the better). Here's how the new version of the `DisplayCustomers()` method has been influenced as a result:

```
[VB code]
Public Sub DisplayCustomers(ByVal customers As List(Of Customer))
    For custIdx As Int32 = 0 To (customers.Count - 1)
        Dim cust As Customer = customers(custIdx)
        Console.Out.WriteLine("Customer-) ID: {0}, Name: {1}", cust.Id, cust.Name)

        Dim orders() As Order = cust.Items
        For orderIdx As Int32 = 0 To (orders.Length - 1)
            Dim ord As Order = orders(orderIdx)
            Console.Out.WriteLine("  Order-> Date: {0}, Item: {1}, Desc: {2}", _
                                  ord.OrderDate, ord.ItemId, ord.Description)
        Next
    Next
End Sub
```

```
[C# code]
public void DisplayCustomers(List<Customer> customers) {
    for (int custIdx = 0; custIdx < customers.Count; custIdx++) {
        Customer cust = customers[custIdx];
        Console.Out.WriteLine("Customer-> ID: {0}, Name: {1}", cust.Id, cust.Name);

        Order[] orders = cust.Items;
        for (int orderIdx = 0; orderIdx < orders.Length; orderIdx++) {
            Order ord = orders[orderIdx];
            Console.Out.WriteLine("  Order-> Date: {0}, Item: {1}, Desc: {2}",
                                    ord.OrderDate, ord.ItemId, ord.Description);
        }
   }
}
```

This type safety work, as you can see, has yielded some nice benefits. Though the code isn't smaller (that wasn't the goal anyway), it is certainly much safer. Gone are the plethora of casts that muddied the prior version of this class.

Casting Consequences

In the previous example, you saw how using the `Object` data type forced the client code to use a series of casts to convert the `Object` to the appropriate data type. This need to cast has a number of implications in terms of the general type safety of your code. Consider the following example:

```
[VB code]
Dim custList As ArrayList = CustomerFinder.GetCustomers()
For idx As Int32 = 0 To (custList.Count - 1)
    Dim cust As Customer = DirectCast(custList(idx), Customer)
Next
```

```
[C# code]
ArrayList custList = CustomerFinder.GetCustomers();
for (int idx = 0; idx < custList.Count; idx++) {
    Customer cust = (Customer)custList[idx];
}
```

Certainly, as discussed earlier, the casts that you see in this example are anything but type-safe. However, there's more wrong here than just the absence of type safety. First, the cast that is applied here will have an impact on performance. Although the added overhead is not large, it could still be significant in scenarios where you might need a tight, high-performing loop.

The larger issue, though, is centered more around the fact that casts may not always succeed. And a failed cast can mean unexpected failures in your application. In this example, this code simply presumes that the collection contains `Customer` objects and that each of these casts never throws an exception. This approach just assumes that, as the code evolves, it will never alter the representation of the objects returned by this `GetCustomers()` call. Creating this blind-faith, implied contract between a client and method is dangerous and prone to generating unexpected errors.

You can attempt to manage this through exception handling. This would be achieved by adding the following exception handling block:

```
[VB code]
Try
    Dim custList As ArrayList = CustomerFinder.GetCustomers()
    For idx As Int32 = 0 To (custList.Count - 1)
        Dim cust As Customer = DirectCast(custList(idx), Customer)
    Next
Catch ex As InvalidCastException
    Console.Out.WriteLine(ex.Message)
End Try
```

```
[C# code]
try {
    ArrayList custList = CustomerFinder.GetCustomers();
    for (int idx = 0; idx < custList.Count; idx++) {
        Customer cust = (Customer)custList[idx];
    }
} catch (InvalidCastException ex) {
    Console.Out.WriteLine(ex.Message);
}
```

This modification ensures that you'll catch the casting errors. This is certainly the appropriate action to take and will, at minimum, allow you to easily detect when the errors occur. Because there's likely no appropriate action to take in response to this error, it will likely result, in most cases, in some form of hard error. It's really the only option you have.

You might think you could use the `for each` construct to make this problem go away. Suppose you were to change the loop to the following:

```
[VB codes]
For each cust As Customer in custCollection
    . . .
Next
```

```
[C# code]
foreach (Customer cust in custCollection) {
    . . .
}
```

This seems, on the surface, like safer code. After all, it does eliminate the need for a cast. While it would seem as though this solves the problem, it's probably obvious why it really doesn't. Even though you don't explicitly do a cast in this situation, the resulting code still does. So, if your `custCollection` doesn't contain `Customer` objects, it too will yield an `InvalidCastException`. In many respects, this loop actually causes more problems than the prior example. If you happen to capture an exception here, and you want to continue processing additional items, you cannot use the `continue` construct.

This whole idea of trying to adopt a strategy for dealing with the occurrence of `InvalidCastException`s seems like it's focusing on the wrong dimension of the problem. If you weren't forced to use unsafe types, you wouldn't be in a position of having to coerce them to another type with the hope that the conversion

is successful. Although I'm not saying casting should be eliminated, I am saying it's something you should attempt to avoid.

Fortunately, with generics, this entire discussion is moot. There wouldn't be any casting in these examples if they leveraged generics and, therefore, there won't be any need to worry about strategies for dealing with failed casts (at least in this scenario).

Interface Type Safety

An interface lets you define a signature for a type entirely separate from any implementation of that type. And, because a number of classes might be implementing your interfaces, you should be especially diligent about ensuring their type safety. To clarify this point, let's start by looking at the `ICloneable` interface you may have already been using:

```
[VB code]
Public Interface ICloneable
    Function Clone() As Object
End Interface

[C# code]
Interace ICloneable {
    Public object Clone();
}
```

By now, it should be clear that there's absolutely nothing type-safe about this interface. Any class that implements this interface is free to return any object type in its implementation of the `Clone()` method. Once again, each client is left to their own devices to figure out how to handle the possible fallout of an invalid type being returned from this method.

So, as you can imagine, interfaces are one of the most natural places to leverage the benefits of generics. With one minor modification, this once type-unfriendly interface can become fully type-safe. The generic version would appear as follows:

```
[VB code]
Public Interface ICloneable(Of T)
    Function Clone() As T
End Interface

[C# code]
Interace ICloneable<T> {
    Public T Clone();
}
```

Each class that implements this interface will be required to return a type `T` from its `Clone()` method. If any code attempts to return any other type, the compiler will now capture this condition and throw an error — a much better alternative than entrusting your safety to run-time detection of type collisions.

Scratching the Surface

The preceding examples represent just a few of the countless permutations of how the type safety of your solutions can be improved through the application of generics. The goal here isn't to point out every way generics can be leveraged to improve the type safety of your code. Instead, the idea here is to simply scratch the generic surface enough to expose the impact generics can have on the general type safety of your code.

Once you get in this mindset, you'll find yourself looking at your interfaces in a new light. As you do, you'll find that generics actually provide solutions for a broad spectrum of issues, including interfaces, methods, delegates, classes, and so on. Ultimately, you should find yourself wondering why generics weren't part of the language sooner. If you're in that camp, you're going to have a greater appreciation for the value of generics and are likely to see the more global implications of applying generics to your existing solutions.

Safety vs. Clarity

There's a lot of debate in the .NET development community about the influence of generics on the readability of code. Some view the introduction of generics as an abomination that muddies the syntactic qualities of each language they touch. I find this perspective puzzling. I don't know if this is rooted in the complexity of C++ templates or if the objection is made on some more general basis. Whatever the reason, I still have trouble understanding the fundamental logic behind this mindset.

Although generics add some verbosity to your code, that very verbosity is what enables generics to bring clarity to your code. Consider these two contrasting examples:

```
[VB code]
Public Function FindCustomers(searchParams As HashTable) As ArrayList

Public Function FindCustomers(searchParams As Dictionary(Of string, Int32)) _
                                                          As List(Of Customer)
```

```
[C# code]
public ArrayList FindCustomers(ArrayList searchParams);

public List<Customer> FindCustomers(Dictionary<string, int> searchParams);
```

This example includes non-generic and generic versions of a `FindCustomers()` method. Although the non-generic version is certainly shorter than its generic counterpart, it tells you nothing about the types required for the incoming parameters or the type of objects being returned. If you put aside the obvious type-safety problems here and focus solely on the expressive qualities of these two declarations, you'd have to favor the generic version. Its signature tells you precisely what data types are used for your incoming key/value parameter pairs. It also is very explicit about the type of objects that will be held in the returned list.

So, when I look at these two examples, I see the added syntax introduced by generics as a blessing. I don't see it as muddying the profile of my method. Instead, I see it as adding a much-needed means of qualifying, in detail, the nature of my types.

The truth is, generics should allow you to demand much more from the APIs you consume and expose. When an API hands you back an `ArrayList`, what is it really telling you? It's as if it's saying: "Here's a collection of objects; now you go figure out what it contains." It then becomes your job to track down, sometimes through multiple levels of indirection, the code that created and populated the `ArrayList` to determine what it contains. You are then forced to couple, through casting or some other mechanism, your code to the types contained in the collection with the expectation that the provider of the collection won't change its underlying representation. This whole mechanism of passing out untyped parameters and then binding to their representations creates a level of indirect coupling that can end up being both error-prone and a maintenance headache.

When I look at an interface, I don't want there to be any ambiguity about what it accepts or what it returns. There shouldn't be room for interpretation. Through generics, you are provided with new tools that can make your interfaces much more expressive. And, although this expressiveness makes the syntax more verbose and may rarely make your code run faster, it should still represent a significant factor in measuring the quality of your code.

As developers get more comfortable with generics, any objections to the syntactic impact of this new language feature are likely to subside. The benefits they bring to your code are simply too significant to be brushed aside simply because they tend to increase the verbosity of your declarations.

Summary

Type safety is one of the key value propositions of generics. As such, it is vital for you to have a good grasp on how generics can be applied in ways that will enhance the overall type safety of your solutions. The goal of this chapter was to try and expose some of the type-safety compromises developers have been traditionally forced to make and discuss how generics can be employed to remedy these problems. The chapter looked at how types have been required to use least common denominator object types to achieve some level of generality and, in doing so, accept the overhead and safety issues that accompany that approach. As part of exploring these type-safety issues, the chapter also looked at how generics could be applied to eliminate a great deal of these type-safety problems. You also learned how generics bring a new level of expressiveness to your code and how generics can improve the quality and maintainability of your solutions. Overall, the chapter should give you a real flavor for how generics will influence the expectations you place on the signatures of the types you create and consume.