13a

Code Contracts

Introduced in Framework 4.0, *code contracts* allow methods to interact through a set of mutual obligations, and fail *early* if those obligations are violated.

The types in this chapter are defined primarily in the `System.Diagnostics` and `System.Diagnostics.Contracts` namespaces.

Overview

We mentioned in Chapter 13 of *C# 7.0 in a Nutshell*, the concept of an *assertion*, whereby you check that certain conditions are met throughout your program. If a condition fails, it indicates a bug, which is typically handled by invoking a debugger (in debug builds) or throwing an exception (in release builds).

Assertions follow the principle that if something goes wrong, it's best to fail early and close to the source of the error. This is usually better than trying to continue with invalid data—which can result in incorrect results, undesired side-effects, or an exception later on in the program (all of which are harder to diagnose).

Historically, there have been two ways to enforce assertions:

- By calling the `Assert` method on `Debug` or `Trace`
- By throwing exceptions (such as `ArgumentNullException`)

Code contracts replaces both of these approaches with a unified system that allows you to make not only simple assertions but also more powerful *contract*-based assertions.

Code contracts derive from the principle of "Design by Contract" from the Eiffel programming language, where functions interact with each other through a system of mutual obligations and benefits. Essentially, a function specifies *preconditions* that must be met by the client (caller), and in return guarantees *postconditions* which the client can depend on when the function returns.

The types for code contracts live in the `System.Diagnostics.Contracts` namespace.

> Although the types that support code contracts are built into the .NET Framework, the binary rewriter and the static checking tools are available as a separate download at the Microsoft DevLabs site (*http://msdn.microsoft.com/devlabs*). You must install these tools before you can use code contracts in Visual Studio.

Why Use Code Contracts?

To illustrate, we'll write a method that adds an item to a list only if it's not already present—with two *preconditions* and a *postcondition*:

```
public static bool AddIfNotPresent<T> (IList<T> list, T item)
{
  Contract.Requires (list != null);          // Precondition
  Contract.Requires (!list.IsReadOnly);      // Precondition
  Contract.Ensures (list.Contains (item));   // Postcondition
  if (list.Contains(item)) return false;
  list.Add (item);
  return true;
}
```

The preconditions are defined by `Contract.Requires` and are verified when the method starts. The postcondition is defined by `Contract.Ensures` and is verified not where it appears in the code, but *when the method exits*.

Preconditions and postconditions act like assertions and, in this case, detect the following errors:

- Calling the method with a null or read-only list
- A bug in the method whereby we forgot to add the item to the list

> Preconditions and postconditions must appear at the start of the method. This is conducive to good design: if you fail to fulfill the contract in subsequently writing the method, the error will be detected.

Moreover, these conditions form a discoverable *contract* for that method. `AddIfNotPresent` advertises to consumers:

- "You must call me with a non-null writable list."
- "When I return, that list will contain the item you specified."

These facts can be emitted into the assembly's XML documentation file (you can do this in Visual Studio by going to the Code Contracts tab of the Project Properties window, enabling the building of a contracts reference assembly, and checking "Emit Contracts into XML doc file"). Tools such as SandCastle can then incorporate contract details into documentation files.

Contracts also enable your program to be analyzed for correctness by static contract validation tools. If you try to call `AddIfNotPresent` with a `list` whose value might be null, for example, a static validation tool could warn you before you even run the program.

Another benefit of contracts is ease of use. In our example, it's easier to code the postcondition upfront than at both exit points. Contracts also support *object invariants*—which further reduce repetitive coding and make for more reliable enforcement.

Conditions can also be placed on interface members and abstract methods, something that is impossible with standard validation approaches. And conditions on virtual methods cannot be accidentally circumvented by subclasses.

Yet another benefit of code contracts is that contract violation behavior can be customized easily and in more ways than if you rely on calling `Debug.Assert` or throwing exceptions. And it's possible to ensure that contract violations are always recorded—even if contract violation exceptions are swallowed by exception handlers higher in the call stack.

The disadvantage of using code contracts is that the .NET implementation relies on a *binary rewriter*—a tool that mutates the assembly after compilation. This slows the build process, as well as complicating services that rely on calling the C# compiler (whether explicitly or via the `CSharpCodeProvider` class).

The enforcing of code contracts may also incur a runtime performance hit, although this is easily mitigated by scaling back contract checking in release builds.

> Another limitation of code contracts is that you can't use them to enforce security-sensitive checks, because they can be circumvented at runtime (by handling the `ContractFailed` event).

Contract Principles

Code contracts comprise *preconditions*, *postconditions*, *assertions*, and *object invariants*. These are all discoverable assertions. They differ based on when they are verified:

- *Preconditions* are verified when a function starts.
- *Postconditions* are verified before a function exits.
- *Assertions* are verified wherever they appear in the code.
- *Object invariants* are verified after every public function in a class.

Code contracts are defined entirely by calling (static) methods in the `Contract` class. This makes contracts *language-independent*.

Contracts can appear not only in methods, but in other functions as well, such as constructors, properties, indexers, and operators.

Compilation

Almost all methods in the `Contract` class are defined with the `[Conditional("CONTRACTS_FULL")]` attribute. This means that unless you define the `CONTRACTS_FULL` symbol, (most) contract code is stripped out. Visual Studio defines the `CONTRACTS_FULL` symbol automatically if you enable contract checking in the Code Contracts tab of the Project Properties page. (For this tab to appear, you must download and install the Contracts tools from the Microsoft DevLabs site.)

> Removing the `CONTRACTS_FULL` symbol might seem like an easy way to disable all contract checking. However, it doesn't apply to `Requires<TException>` conditions (which we'll describe in detail soon).
>
> The only way to disable contracts in code that uses `Requires<TException>` is to enable the `CONTRACTS_FULL` symbol and then get the binary rewriter to strip out contract code by choosing an enforcement level of "none."

The binary rewriter

After compiling code that contains contracts, you must call the binary rewriter tool, *ccrewrite.exe* (Visual Studio does this automatically if contract checking is enabled). The binary rewriter moves postconditions (and object invariants) into the right place, calls any conditions and object invariants in overridden methods, and replaces calls to `Contract` with calls to a *contracts runtime class*. Here's a (simplified) version of what our earlier example would look like after rewriting:

```
static bool AddIfNotPresent<T> (IList<T> list, T item)
{
  __ContractsRuntime.Requires (list != null);
  __ContractsRuntime.Requires (!list.IsReadOnly);
  bool result;
  if (list.Contains (item))
    result = false;
  else
  {
    list.Add (item);
    result = true;
  }
  __ContractsRuntime.Ensures (list.Contains (item));   // Postcondition
  return result;
}
```

If you fail to call the binary rewriter, `Contract` won't get replaced with `__ContractsRuntime` and the former will end up throwing exceptions.

> The `__ContractsRuntime` type is the default contracts runtime class. In advanced scenarios, you can specify your own contracts runtime class via the `/rw` switch or Visual Studio's Code Contracts tab in Project Properties.
>
> Because `__ContractsRuntime` is shipped with the binary rewriter (which is not a standard part of the .NET Framework), the binary rewriter actually injects the `__ContractsRuntime` class into your compiled assembly. You can examine its code by disassembling any assembly that enables code contracts.

The binary rewriter also offers switches to strip away some or all contract checking: we describe these in "Selectively Enforcing Contracts." You typically enable full contract checking in debug build configurations and a subset of contract checking in release configurations.

Asserting versus throwing on failure

The binary rewriter also lets you choose between displaying a dialog and throwing a `ContractException` upon contract failure. The former is typically used for debug builds; the latter for release builds. To enable the latter, specify `/throwonfailure` when calling the binary rewriter, or uncheck the "Assert on contract failure" checkbox in Visual Studio's Code Contracts tab in Project Properties.

We'll revisit this topic in more detail in "Dealing with Contract Failure."

Purity

All functions that you call from arguments passed to contract methods (`Requires`, `Assumes`, `Assert`, etc.) must be *pure*—that is, side-effect-free (they must not alter the values of fields). You must signal to the binary rewriter that any functions you call are pure by applying the `[Pure]` attribute:

```
[Pure]
public static bool IsValidUri (string uri) { ... }
```

This makes the following legal:

```
Contract.Requires (IsValidUri (uri));
```

The contract tools implicitly assume that all property get accessors are pure, as are all C# operators (`+`, `*`, `%`, etc.) and members on selected Framework types, including `string`, `Contract`, `Type`, `System.IO.Path`, and LINQ's query operators. It also assumes that methods invoked via delegates marked with the `[Pure]` attribute are pure (the `Comparison<T>` and `Predicate<T>` attributes are marked with this attribute).

Preconditions

You can define code contract preconditions by calling `Contract.Requires`, `Contract.Requires<TException>` or `Contract.EndContractBlock`.

Contract.Requires

Calling `Contract.Requires` at the start of a function enforces a precondition:

```
static string ToProperCase (string s)
{
  Contract.Requires (!string.IsNullOrEmpty(s));
  ...
}
```

This is like making an assertion, except that the precondition forms a discoverable fact about your function that can be extracted from the compiled code and consumed by documentation or static checking tools (so that they can warn you should they see some code elsewhere in your program that tries to call `ToProperCase` with a null or empty string).

A further benefit of preconditions is that subclasses that override virtual methods with preconditions cannot prevent the base class method's preconditions from being checked. And preconditions defined on *interface*

members will be implicitly woven into the concrete implementations (see "Contracts on Interfaces and Abstract Methods").

> Preconditions should access only members that are at least as accessible as the function itself—this ensures that callers can make sense of the contract. If you need to read or call less accessible members, it's likely that you're validating *internal state* rather than enforcing the *calling contract*, in which case you should make an assertion instead.

You can call `Contract.Requires` as many times as necessary at the start of the method to enforce different conditions.

What Should You Put in Preconditions?

The guideline from the Code Contracts team is that preconditions should:

- Be possible for the client (caller) to easily validate.
- Rely only on data & functions at least as accessible as the method itself.
- Always indicate a *bug* if violated.

A consequence of the last point is that a client should never specifically "catch" a contract failure (the `ContractException` type, in fact, is internal to help enforce that principle). Instead, the client should call the target properly; if it fails, this indicates a bug that should be handled via your general exception backstop (which may include terminating the application). In other words, if you decide control-flow based on a precondition failure, it's not really a contract because you can continue executing if it fails.

This leads to the following advice, when choosing between preconditions vs throwing exceptions:

- If failure *always* indicates a bug in the client, favor a precondition.
- If failure indicates an *abnormal condition*, which *may* mean a bug in the client, throw a (catchable) exception instead.

To illustrate, suppose we're writing the `Int32.Parse` function. It's reasonable to assume that a null input string always indicates a bug in the caller, so we'd enforce this with a precondition:

```
public static int Parse (string s)
{
  Contract.Requires (s != null);
  ...
```

Next, we need to check that the string contains only digits and symbols such as `+` and `-` (in the right place). It would place an unreasonable burden on the caller to validate this, and so we'd enforce it not as a precondition, but a manual check that throws a (catchable) `FormatException` if violated.

To illustrate the member accessibility issue, consider the following code, which often appears in types implementing the `IDisposable` interface:

```
public void Foo()
{
  if (_isDisposed)  // _isDisposed is a private field
    throw new ObjectDisposedException ("...");
  ...
```

This check should not be made into a precondition unless we make `_isDisposed` accessible to the caller (by refactoring it into a publicly readable property, for instance).

Finally, consider the `File.ReadAllText` method. The following would be *inappropriate* use of a precondition:

```
public static string ReadAllText (string path)
{
  Contract.Requires (File.Exists (path));
  ...
```

The caller cannot reliably know that the file exists before calling this method (it could be deleted between making that check and calling the method). So, we'd enforce this in the old-fashioned way—by throwing a catchable `FileNotFoundException` instead.

Contract.Requires<TException>

The introduction of code contracts challenges the following deeply entrenched pattern established in the .NET Framework from version 1.0:

```
static void SetProgress (string message, int percent)  // Classic approach
{
  if (message == null)
    throw new ArgumentNullException ("message");

  if (percent < 0 || percent > 100)
    throw new ArgumentOutOfRangeException ("percent");
  ...
}

static void SetProgress (string message, int percent)  // Modern approach
{
  Contract.Requires (message != null);
  Contract.Requires (percent >= 0 && percent <= 100);
  ...
}
```

If you have a large assembly that enforces classic argument checking, writing new methods with preconditions will create an inconsistent library: some methods will throw argument exceptions whereas others will throw a `ContractException`. One solution is to update all existing methods to use contracts, but this has two problems:

- It's time-consuming.
- Callers may have come to *depend* on an exception type such as `ArgumentNullException` being thrown. (This almost certainly indicates bad design, but may be the reality nonetheless.)

The solution is to call the generic version of `Contract.Requires`. This lets you specify an exception type to throw upon failure:

```
Contract.Requires<ArgumentNullException> (message != null, "message");
Contract.Requires<ArgumentOutOfRangeException>
 (percent >= 0 && percent <= 100, "percent");
```

(The second argument gets passed to the constructor of the exception class).

This results in the same behavior as with old-fashioned argument checking, while delivering the benefits of contracts (conciseness, support for interfaces, implicit documentation, static checking and runtime customization).

> The specified exception is thrown only if you specify `/throwonfailure` when rewriting the assembly (or *uncheck* the *Assert on Contract Failure* checkbox in Visual Studio). Otherwise, a dialog box appears.

It's also possible to specify a contract-checking level of *ReleaseRequires* in the binary rewriter (see "Selectively Enforcing Contracts"). Calls to the generic `Contract.Requires<TException>` then remain in place while all other checks are stripped away: this results in an assembly that behaves just as in the past.

Contract.EndContractBlock

The `Contract.EndContractBlock` method lets you get the benefit of code contracts with traditional argument-checking code—avoiding the need to refactor code written prior to Framework 4.0. All you do is call this method after performing manual argument checks:

```
static void Foo (string name)
{
  if (name == null) throw new ArgumentNullException ("name");
  Contract.EndContractBlock();
  ...
}
```

The binary rewriter then converts this code into something equivalent to:

```
static void Foo (string name)
{
  Contract.Requires<ArgumentNullException> (name != null, "name");
  ...
}
```

The code that precedes `EndContractBlock` must comprise simple statements of the form:

```
if <condition> throw <expression>;
```

You can mix traditional argument checking with code contract calls: simply put the latter after the former:

```
static void Foo (string name)
{
  if (name == null) throw new ArgumentNullException ("name");
  Contract.Requires (name.Length >= 2);
  ...
}
```

Calling any of the contract-enforcing methods implicitly ends the contract block.

The point is to define a region at the beginning of the method where the contract rewriter knows that every `if` statement is part of a contract. Calling any of the contract-enforcing methods implicitly extends the contract block, so you don't need to use `EndContractBlock` if you use another method such as `Contract.Ensures`.

Preconditions and Overridden Methods

When overriding a virtual method, you cannot add preconditions, because doing so would *change the contract* (by making it more restrictive)—breaking the principles of polymorphism.

(Technically, the designers could have allowed overridden methods to *weaken* preconditions; they decided against this because the scenarios weren't sufficiently compelling to justify adding this complexity).

> The binary rewriter ensures that a base method's preconditions are always enforced in subclasses—whether or not the overridden method calls the base method.

Postconditions

Contract.Ensures

`Contract.Ensures` enforces a postcondition: something which must be true when the method exits. We saw an example earlier:

```
static bool AddIfNotPresent<T> (IList<T> list, T item)
{
  Contract.Requires (list != null);          // Precondition
  Contract.Ensures (list.Contains (item));   // Postcondition
  if (list.Contains(item)) return false;
  list.Add (item);
  return true;
}
```

The binary rewriter moves postconditions to the exit points of the method. Postconditions are checked if you return early from a method (as in this example)—but not if you return early via an unhandled exception.

Unlike preconditions, which detect misuse by the *caller*, postconditions detect an error in the function itself (rather like assertions). Therefore, postconditions may access private state (subject to the caveat stated shortly, in "Postconditions and overridden methods").

Postconditions and Thread Safety

Multithreaded scenarios (Chapter 14) challenge the usefulness of postconditions. For instance, suppose we wrote a thread-safe wrapper for a `List<T>` with a method as follows:

```
public class ThreadSafeList<T>
{
  List<T> _list = new List<T>();
  object _locker = new object();

  public bool AddIfNotPresent (T item)
  {
    Contract.Ensures (_list.Contains (item));
    lock (_locker)
    {
      if (_list.Contains(item)) return false;
      _list.Add (item);
      return true;
    }
  }

  public void Remove (T item)
  {
    lock (_locker)
      _list.Remove (item);
  }
}
```

The postcondition in the `AddIfNotPresent` method is checked *after* the lock is released—at which point the item may no longer exist in the list if another thread called `Remove` right then. There is currently no workaround for this problem, other than to enforce such conditions as assertions (see next section) rather than postconditions.

Contract.EnsuresOnThrow<TException>

Occasionally, it's useful to ensure that a certain condition is true should a particular type of exception be thrown. The `EnsuresOnThrow` method does exactly this:

```
Contract.EnsuresOnThrow<WebException> (this.ErrorMessage != null);
```

Contract.Result<T> and Contract.ValueAtReturn<T>

Because postconditions are not evaluated until a function ends, it's reasonable to want to access the return value of a method. The `Contract.Result<T>` method does exactly that:

```
Random _random = new Random();
int GetOddRandomNumber()
{
  Contract.Ensures (Contract.Result<int>() % 2 == 1);
  return _random.Next (100) * 2 + 1;
}
```

The `Contract.ValueAtReturn<T>` method fulfills the same function—but for `ref` and `out` parameters.

Contract.OldValue<T>

`Contract.OldValue<T>` returns the original value of a method parameter. This is useful with postconditions because the latter are checked at the *end* of a function. Therefore, any expressions in postconditions that incorporate parameters will read the *modified* parameter values.

For example, the postcondition in the following method will always fail:

```
static string Middle (string s)
{
  Contract.Requires (s != null && s.Length >= 2);
  Contract.Ensures (Contract.Result<string>().Length < s.Length);
  s = s.Substring (1, s.Length - 2);
  return s.Trim();
}
```

Here's how we can correct it:

```
static string Middle (string s)
{
  Contract.Requires (s != null && s.Length >= 2);
  Contract.Ensures (Contract.Result<string>().Length <
                    Contract.OldValue (s).Length);
  s = s.Substring (1, s.Length - 2);
  return s.Trim();
}
```

Postconditions and Overridden Methods

An overridden method cannot circumvent postconditions defined by its base, but it can add new ones. The binary rewriter ensures that a base method's postconditions are always checked—even if the overridden method doesn't call the base implementation.

> For the reason just stated, postconditions on virtual methods should not access private members. Doing so will result in the binary rewriter weaving code into the subclass that will try to access private members in the base class—causing a runtime error.

Assertions and Object Invariants

In addition to preconditions and postconditions, the code contracts API lets you make assertions and define *object invariants*.

Assertions

Contract.Assert

You can make assertions anywhere in a function by calling `Contract.Assert`. You can optionally specify an error message if the assertion fails:

```
...
int x = 3;
...
Contract.Assert (x == 3);                 // Fail unless x is 3
Contract.Assert (x == 3, "x must be 3");
...
```

The binary rewriter doesn't move assertions around. There are two reasons for favoring `Contract.Assert` over `Debug.Assert`:

- You can leverage the more flexible failure-handling mechanisms offered by code contracts
- Static checking tools can attempt to validate `Contract.Assert`s

Contract.Assume

`Contract.Assume` behaves exactly like `Contract.Assert` at run-time, but has slightly different implications for static checking tools. Essentially, static checking tools won't *challenge* an assumption, whereas they may challenge an assertion. This is useful in that there will always be things a static checker is unable to prove, and this may lead to it "crying wolf" over a valid assertion. Changing the assertion to an assumption keeps the static checker quiet.

Object Invariants

For a class, you can specify one or more *object invariant* methods. These methods run automatically after every *public* function in the class, and allow you to assert that the object is in an internally consistent state.

> Support for multiple object invariant methods was included to make object invariants work well with partial classes.

To define an object invariant method, write a parameterless void method and annotate it with the `[ContractInvariantMethod]` attribute. In that method, call `Contract.Invariant` to enforce each condition that should hold true:

```
class Test
{
  int _x, _y;

  [ContractInvariantMethod]
  void ObjectInvariant()
  {
    Contract.Invariant (_x >= 0);
    Contract.Invariant (_y >= _x);
  }

  public int X { get { return _x; } set { _x = value; } }
  public void Test1() { _x = -3; }
  void Test2()        { _x = -3; }
}
```

The binary rewriter translates the `X` property, `Test1` method and `Test2` method to something equivalent to this:

```
public void X { get { return _x; } set { _x = value; ObjectInvariant(); } }
public void Test1() { _x = -3; ObjectInvariant(); }
void Test2()        { _x = -3; }    // No change because it's private
```

> Object invariants don't *prevent* an object from entering an invalid state: they merely *detect* when that condition has occurred.

`Contract.Invariant` is rather like `Contract.Assert`, except that it can appear only in a method marked with the `[ContractInvariantMethod]` attribute. And conversely, a contract invariant method can only contain calls to `Contract.Invariant`.

A subclass can introduce its own object invariant method, too, and this will be checked in addition to the base class's invariant method. The caveat, of course, is that the check will take place only after a public method is called.

Contracts on Interfaces and Abstract Methods

A powerful feature of code contracts is that you can attach conditions to interface members and abstract methods. The binary rewriter then automatically weaves these conditions into the members' concrete implementations.

A special mechanism lets specify a separate contract class for interfaces and abstract methods, so that you can write method bodies to house the contract conditions. Here's how it works:

```
[ContractClass (typeof (ContractForITest))]
interface ITest
{
  int Process (string s);
}
```

```
[ContractClassFor (typeof (ITest))]
sealed class ContractForITest : ITest
{
  int ITest.Process (string s)     // Must use explicit implementation.
  {
    Contract.Requires (s != null);
    return 0;                      // Dummy value to satisfy compiler.
  }
}
```

Notice that we had to return a value when implementing `ITest.Process` to satisfy the compiler. The code that returns 0 will not run, however. Instead, the binary rewriter extracts just the conditions from that method, and weaves them into the real implementations of `ITest.Process`. This means that the contract class is never actually instantiated (and any constructors that you write will not execute).

You can assign a temporary variable within the contract block to make it easier to reference other members of the interface. For instance, if our `ITest` interface also defined a `Message` property of type `string`, we could write the following in `ITest.Process`:

```
int ITest.Process (string s)
{
  ITest test = this;
  Contract.Requires (s != test.Message);
  ...
}
```

This is easier than:

```
Contract.Requires (s != ((ITest)this).Message);
```

(Simply using `this.Message` won't work because `Message` must be explicitly implemented.) The process of defining contract classes for abstract classes is exactly the same, except that the contract class should be marked `abstract` instead of `sealed`.

Dealing with Contract Failure

The binary rewriter lets you specify what happens when a contract condition fails, via the `/throwonfailure` switch (or the *Assert on Contract Failure* checkbox in Visual Studio's *Contracts* tab in *Project Properties*).

If you don't specify `/throwonfailure`—or check *Assert on Contract Failure*—a dialog appears upon contract failure, allowing you to abort, debug or ignore the error.

There are a couple of nuances to be aware of:

- If the CLR is hosted (i.e., in SQL Server or Exchange), the host's escalation policy is triggered instead of a dialog appearing.
- Otherwise, if the current process can't pop up a dialog box to the user, `Environment.FailFast` is called.

The dialog is useful in debug builds for a couple of reasons:

- It makes it easy to diagnose and debug contract failures on the spot—without having to re-run the program. This works regardless of whether Visual Studio is configured to break on first-chance exceptions. And unlike with exceptions in general, contract failure almost certainly means a bug in your code.
- It lets you know about contract failure—even if a caller higher up in the stack "swallows" exceptions as follows:

```
try
{
  // Call some method whose contract fails
}
catch { }
```

> The code above is considered an antipattern in most scenarios because it *masks* failures, including conditions that the author never anticipated.

If you specify the `/throwonfailure` switch and uncheck *Assert on Contract Failure* in Visual Studio—a `ContractException` is thrown upon failure. This is desirable for:

- Release builds—where you would let the exception bubble up the stack and be treated like any other unexpected exception (perhaps by having a top-level exception handler log the error or invite the user to report it).
- Unit testing environments— where the process of logging errors is automated.

> `ContractException` cannot appear in a `catch` block because this type is not public. The rationale is that there's no reason that you'd want to *specifically* catch a `ContractException`—you'd want to catch it only as part of a general exception backstop.

The ContractFailed Event

When a contract fails the static `Contract.ContractFailed` event fires before any further action is taken. If you handle this event, you can query the event arguments object for details of the error. You can also call `SetHandled` to prevent a `ContractException` from being subsequently thrown (or a dialog appearing).

Handling this event is particularly useful when `/throwonfailure` is specified, because it lets you log *all* contract failures—even if code higher in the call stack swallows exceptions as we described just before. A great example is with automated unit testing:

```
Contract.ContractFailed += (sender, args) =>
{
  string failureMessage = args.FailureKind + ": " + args.Message;
  // Log failureMessage with unit testing framework:
  // ...
  args.SetUnwind();
};
```

This handler logs all contract failures, while allowing the normal `ContractException` (or contract failure dialog) to run its course after the event handler has finished. Notice that we also call `SetUnwind`: this neutralizes the effect of any calls to `SetHandled` from other event subscribers. In other words, it ensures that a `ContractException` (or dialog) will always follow after all event handlers have run.

If you throw an exception from within this handler, any other event handlers will still execute. The exception that you threw then populates the `InnerException` property of the `ContractException` that's eventually thrown.

Exceptions Within Contract Conditions

If an exception is thrown within a contract condition itself, then that exception propagates like any other—regardless of whether `/throwonfailure` is specified. The following method throws a `NullReferenceException` if called with a null string:

```
string Test (string s)
{
  Contract.Requires (s.Length > 0);
  ...
}
```

This precondition is essentially faulty. It should instead be:

```
Contract.Requires (!string.IsNullOrEmpty (s));
```

Selectively Enforcing Contracts

The binary rewriter offers two switches that strip away some or all contract checking: `/publicsurface` and `/level`. You can control these from Visual Studio via the *Code Contracts* tab of *Project Properties*. The `/publicsurface` switch tells the rewriter to check contracts only on public members. The `/level` switch has the following options:

None (Level 0)
: Strips out *all* contract verification

ReleaseRequires (Level 1)
: Enables only calls to the generic version of `Contract.Requires<TException>`

Preconditions (Level 2)
: Enables all preconditions (Level 1 plus normal preconditions)

Pre and Post (Level 3)
: Enables Level2 checking plus postconditions

Full (Level 4)
: Enables Level 3 checking plus object invariants and assertions (i.e., everything)

You typically enable full contract checking in debug build configurations.

Contracts in Release Builds

When it comes to making release builds, there are two general philosophies:

- Favor safety and enable full contract checking
- Favor performance and disable all contract checking

If you're building a library for public consumption, though, the second approach creates a problem. Imagine that you compile and distribute library L in release mode with contract checking disabled. A client then builds project C in *debug* mode that references library L. Assembly C can then call members of L incorrectly without contract violations! In this situation, you actually want to enforce the parts of L's contract that ensure correct usage of L—in other words, the *preconditions* in L's *public* members.

The simplest way to resolve this is to enable `/publicsurface` checking in L with a level of *Preconditions* or *ReleaseRequires*. This ensures that the essential preconditions are enforced for the benefit of consumers, while incurring the performance cost of only those preconditions.

In extreme cases, you might not want to pay even this small performance price—in which case you can take the more elaborate approach of *call-site checking*.

Call-Site Checking

Call-site checking moves precondition validation from *called* methods into *calling* methods (call sites). This solves the problem just described—by enabling consumers of library L to perform L's precondition validation themselves in debug configurations.

To enable call-site checking, you must first build a separate *contracts reference assembly*—a supplementary assembly that contains just the preconditions for the referenced assembly. To do this, you can either use the *ccrefgen* command-line tool, or proceed in Visual Studio as follows:

1. In the release configuration of the *referenced library* (L), go to the *Code Contracts* tab of *Project Properties* and disable runtime contract checking while ticking "Build a Contract Reference Assembly". This then generates a supplementary contracts reference assembly (with the suffix *.contracts.dll*).
2. In the *release* configuration of the *referencing* assemblies, disable all contract checking.
3. In the *debug* configuration of the *referencing* assemblies, tick "Call-site Requires Checking".

The third step is equivalent to calling *ccrewrite* with the `/callsiterequires` switch. It reads the preconditions from the contracts reference assembly and weaves them into the calling sites in the referencing assembly.

Static Contract Checking

Code contracts make *static contract checking* possible, whereby a tool analyzes contract conditions to find potential bugs in your program before it's run. For example, statically checking the following code generates a warning:

```
static void Main()
{
  string message = null;
  WriteLine (message);     // Static checking tool will generate warning
}

static void WriteLine (string s)
{
  Contract.Requires (s != null);
  Console.WriteLine (s);
}
```

You can run Microsoft's static contracts tool from the command line via *cccheck*, or by enabling static contract checking in Visual Studio's project properties dialog.

For static checking to work, you may need to add preconditions and postconditions to your methods. To give a simple example, the following will generate a warning:

```
static void WriteLine (string s, bool b)
{
  if (b)
    WriteLine (s);    // Warning: requires unproven
}

static void WriteLine (string s)
{
  Contract.Requires (s != null);
  Console.WriteLine (s);
}
```

Because we're calling a method that requires the parameter to be non-null, we must prove that the argument is non-null. To do this, we can add a precondition to the first method as follows:

```
static void WriteLine (string s, bool b)
{
  Contract.Requires (s != null);
  if (b)
    WriteLine (s);    // OK
}
```

The ContractVerification Attribute

Static checking is easiest if instigated from the beginning of a project's lifecycle—otherwise you're likely to get overwhelmed with warnings.

If you do want to apply static contract checking to an existing codebase, it can help by initially applying it just to selective parts of a program—via the `ContractVerification` attribute (in `System.Diagnostics.Contracts`). This attribute can be applied at the assembly, type and member level. If you apply it at multiple levels, the more granular wins. Therefore, to enable static contract verification just for a particular class, start by disabling verification at the assembly-level as follows:

```
[assembly: ContractVerification (false)]
```

and then enable it just for the desired class:

```
[ContractVerification (true)]
class Foo { ... }
```

Baselines

Another tactic in applying static contract verification to an existing codebase is to run the static checker with the *Baseline* option checked in Visual Studio. All the warnings that are produced are then written to a specified XML file. Next time you run static verification, all the warnings in that that file are ignored—so you see only messages generated as a result of *new* code that you've written.

The SuppressMessage Attribute

You can also tell the static checker to ignore certain types of warnings via the `SuppressMessage` attribute (in `System.Diagnostics.CodeAnalysis`):

```
[SuppressMessage ("Microsoft.Contracts", warningFamily)]
```

where *warningFamily* is one of the following values:

```
Requires Ensures Invariant NonNull DivByZero MinValueNegation
ArrayCreation ArrayLowerBound ArrayUpperBound
```

You can apply this attribute at an assembly or type level.