PARTE 1

Apresentando C# e a Plataforma .NET

CAPÍTULO 1

A Filosofia .NET

Em intervalos de alguns anos, o programador moderno sente a necessidade de fazer um transplante de conhecimento, auto-imposto, para ficar em dia com as novas tecnologias da atualidade. Chega um momento em que as linguagens (C++, Visual Basic 6.0, Java), frameworks (OWL, MFC, ATL, STL), arquiteturas (COM, CORBA, EJB), e APIs (como os Formulários Windows .NET e as bibliotecas GDI+) que foram anunciados como as balas de prata do desenvolvimento de software, tornam-se marginalizadas por algo melhor ou, no mínimo, novo. Não importa a frustração que você possa sentir ao atualizar sua base interna de conhecimento, francamente, isto é inevitável. Para isto, o objetivo deste livro é analisar os detalhes da oferta da Microsoft dentro do cenário da engenharia de software: a plataforma .NET e a linguagem de programação C#.

O objetivo deste capítulo é construir uma base para o restante do livro. Aqui, você encontrará uma discussão de alto nível sobre diversos assuntos relacionados a .NET, como assemblies, a linguagem intermediária comum (sigla em inglês: CIL (Commom Intermediate Language) e a compilação justi-in-time (JIT). Além de visualizar alguns recursos importantes da linguagem de programação C#, também poderá compreender a relação entre diversos aspectos do .NET Framework, common language runtime (CLR), o Common Type System (CTS) e o Common Language Specification (CLS).

Este capítulo também oferece uma pesquisa sobre a funcionalidade fornecida pelas bibliotecas de classe de base .NET, às vezes abreviado como BCL, ou como FCL (bibliotecas de classe do Framework). Finalmente, oferece uma visão geral da natureza de ignorância de linguagem e independência de plataforma da plataforma .NET (sim, é verdade, o .NET não é restrito ao sistema operacional Windows). Como poderia esperar, todos estes tópicos serão explorados com detalhes no restante deste texto.

Compreendendo a Situação Anterior

Antes de analisar as especificações do universo .NET, é bom considerar alguns dos assuntos que motivaram a gênese da atual plataforma Microsoft. Para entrarmos no foco correto, vamos começar este capítulo com uma breve e indolor lição de história para lembrar nossas raízes e compreender as limitações da situação anterior (afinal de contas, admitir ter um problema é o primeiro passo para encontrar uma solução). Depois de concluir esta rápida excursão da vida como a conhecíamos, voltaremos nossa atenção aos numerosos benefícios fornecidos pelo C# e a plataforma.NET.

A Vida Como um Programador de C/Win32 API

Tradicionalmente falando, desenvolver software para a família de sistemas operacionais Windows envolvia a linguagem de programação C em conjunto com a interface de programa aplicativo Windows (API). Enquanto é verdade que diversas aplicações foram criadas com sucesso através desta abordagem respeitável, poucos de nós discordariam que construir aplicações utilizando API pura é um empreendimento complexo.

O primeiro problema óbvio é que C é uma linguagem muito concisa. Os desenvolvedores em C são forçados a debater com o gerenciamento manual de memória, operadores aritméticos e construções sintáticas horríveis. Além disso, como C é uma linguagem estruturada, ela não aproveita os benefícios fornecidos pela abordagem orientada a objetos (alguém diria *código espaguete?*). Ao combinar milhares de funções globais e tipos de dados definidos pela Win32 API em uma já formidável linguagem, não é difícil concluir que haverá tantas aplicações com bug por

aí atualmente.

A Vida Como um Programador de C++/MFC

Uma grande melhoria sobre o desenvolvimento puro em C/API é a utilização da linguagem de programação C++. De muitas maneiras, pode-se pensar que o C++ é uma *camada* orientada a objeto sobre o C. Portanto, embora os programadores de C++ tirem proveito dos famosos "pilares do OOP" (encapsulamento, herança e polimorfismo), eles continuam à mercê dos aspectos dolorosos da linguagem C (isto é, gerenciamento manual de memória, operadores aritméticos e construções sintáticas horríveis).

Apesar de sua complexidade, existem muitos frameworks C++. Por exemplo: a Microsoft Foundation Classes (MFC) fornece ao desenvolvedor um conjunto de classes C++ que facilita a construção de aplicações Win32. O papel principal da MFC é empacotar um "subconjunto são" da API Win32 pura, por trás de diversas classes, macros mágicas e diversas ferramentas de geração de código (mais conhecidos como assistentes). Não importa a assistência oferecida pelo framework MFC (bem como de outros kits de ferramentas baseados em C++). A questão central é que a programação em C++ continua sendo uma experiência difícil e com tendência a erros devido às suas raízes históricas em C.

A Vida Como um Programador de Visual Basic 6.0

Devido a um desejo sincero de levar um estilo de vida mais simples, muitos programadores desviram-se do mundo dos frameworks baseados em C(++) para ingressarem em linguagens mais gentis, como o 6.0 (VB6). O VB6 é popular devido à sua capacidade de construir interfaces de usuário complexas, bibliotecas de código (por exemplo: servidores COM) e lógica de acesso de dados com um mínimo de bagunça e preocupação. Ainda mais que a MFC, o VB6 oculta as complexidades da API Win32 pura, utilizando diversos assistentes de código integrado, tipos de dados intrínsecos, classes e funções específicas do VB.

A principal desvantagem do VB6 (que foi corrigida com o advento da plataforma .NET) é que não se trata de uma linguagem totalmente orientada a objeto, mas sim de uma linguagem "compatível com objetos". Por exemplo: o VB6 não permite que o programador estabeleça relações "is-a" (é um) entre tipos (isto é, não há herança clássica) e não tem suporte intrínseco para construção parametrizada de classe. Além do mais, o VB6 não fornece a capacidade de desenvolver aplicações multitarefa a não ser que queira descer ao nível dos chamados API Win32 (que são, no mínimo, complexos e, no máximo, perigosos).

A Vida Como um Programador de Java/J2EE

Bem-vindo ao Java, uma linguagem de programação orientada a objeto com raízes sintáticas em C++. Como muitas pessoas sabem, a potência do Java é muito maior do que seu suporte à independência de plataforma. O Java (como linguagem) elimina muitos aspectos sintáticos ruins do C++. Já como plataforma, fornece aos programadores um número grande de "pacotes" pré-definidos, que contêm varias definições de tipos. Utilizando estes tipos, os programadores de Java conseguem construir aplicações "Java 100% Puras" com conectividade a base de dados, suporte a mensagens, front-ends habilitados à web e uma interface de usuário de desktop rica.

Embora Java seja uma linguagem elegante, um problema potencial é que sua utilização tipicamente significa que deve usá-la do início ao fim durante o ciclo de desenvolvimento. Na verdade, representa pouca esperança em integração de linguagem, já que isto vai contra o objetivo principal do Java (uma única linguagem de programação para todas as necessidades). Na realidade, entretanto, existem milhões de linhas de código existente no mundo que, idealmente, gostariam de se fundir ao código Java mais novo. Infelizmente, esta linguagem dificulta esta tarefa. Enquanto o Java fornece uma capacidade limitada para acessar APIs não-Java, existe muito pouco suporte para a verdadeira integração entre linguagens.

A Vida Como um Desenvolvedor COM

O Component Object Model (COM) era o antigo framework de desenvolvimento de aplicações da Microsoft. COM é uma arquitetura que diz, com efeito: "Se você construir suas classes de acordo com as regras do COM, acabará com um bloco de *código binário reutilizável*."

A beleza de um servidor COM binário é que ele pode ser acessado de maneira independente

da linguagem. Portanto, os programadores de C++ podem construir classes COM que podem ser usadas pelo VB6. Programadores de Delphi podem usar a construção de classes COM utilizando C, e assim por diante. Entretanto, como deve saber, a independência da linguagem COM é um tanto quanto limitada. Por exemplo: não há como derivar uma nova classe COM utilizando uma classe COM existente (o COM não suporta herança clássica). Uma alternativa é utilizar a relação mais complexa "has-a" (tem um) para reutilizar tipos de classe COM.

Outro benefício do COM é sua natureza de transparência local. Utilizando estruturas como identificadores de aplicação (AppIDs), stubs, proxies e o ambiente de tempo de execução do COM, os programadores podem evitar a necessidade de trabalhar com soquetes brutos, chamadas RPC e outros detalhes menos importantes. Por exemplo: considere o código de cliente VB6 COM a seguir:

```
"O tipo MyCOMClass poderia ser escrito em qualquer linguagem
compatível com COM, e poderia ser inserido em qualquer ponto da
rede (inclusive em sua máquina local).
objeto Dim como MyCOMClass
Set obj = New MyCOMClass 'Local resolvido através de AppID.
obj.DoSomeWork
```

Embora o COM possa ser considerado um modelo de objeto muito bem sucedido, é extremamente sensível por trás de sua aparência (pelo menos até você ter passado vários meses explorando seus encanamentos – principalmente se é um programador de C++). Para ajudar a simplificar o desenvolvimento de binários COM, diversos frameworks compatíveis com COM passaram a existir. Por exemplo: a Active Template Library (ATL) fornece outro conjunto de classes C++, templates e macros para facilitar a criação de tipos COM.

Muitas outras linguagens também ocultam uma boa parte da infra-estrutura COM. Entretanto, apenas o suporte à linguagem não é suficiente para ocultar a complexidade do COM. Mesmo quando escolhe uma linguagem compatível com COM relativamente simples, como VB6, ainda é forçado a usar conteúdo com estradas de registro frágeis e diversos problemas relacionados à implementação (coletivamente e, de certa forma, comicamente, chamados de *inferno DLL*).

A Vida Como um Programador de Windows DNA

Para complicar ainda mais, existe uma coisinha chamada Internet. Ao longo dos últimos anos, a Microsoft adicionou mais recursos compatíveis com a Internet à sua família de sistemas operacionais e produtos. Infelizmente, desenvolver uma aplicação Web utilizando a Windows Distributed interNet Applications Architecture (DNA) baseada em COM também é bastante complexo.

Um pouco desta complexidade deve-se ao simples fato de que a Windows DNA exige a utilização de diversas tecnologias e linguagens (ASP, HTML, XML, JScript, VBScript e COM[+], bem como acesso de dados API, como ADO). Um problema é que muitas destas tecnologias não se relacionam de jeito nenhum do ponto de vista sintático. Por exemplo: JScript tem uma sintaxe parecida com C, enquanto VBScript é um subconjunto do VB6. Os servidores COM criados para rodar sob o tempo de execução COM+ têm uma aparência e uma sensação totalmente diferentes das páginas ASP que os invocam. O resultado é uma combinação altamente confusa de tecnologias.

Além disso, e talvez, mais importante, cada linguagem e/ou tecnologia tem seu próprio sistema de tipos (que pode não parecer em nada com o outro sistema de tipos). Além do fato de cada API vir com sua própria coleção de código pré-fabricado, até mesmo os tipos de dados básicos, nem sempre podem ser tratados de forma idêntica. Um `CComBSTR` em ATL não é a mesma coisa que um `String` em VB6, e ambos não têm nada a ver com um `char*` em C.

A Solução .NET

Quanta coisa para uma breve lição de história. A questão é que a vida como programador Windows tem sido dura. O .NET Framework é uma abordagem bastante radical e bruta para facilitar nossas vidas. A solução proposta pelo .NET é "Mudar tudo" (desculpe, não se pode culpar o mensageiro pela mensagem). Como verá no restante deste livro, o .NET Framework é um modelo completamente novo para construir sistemas na família Windows de sistemas operacionais, bem como em diversos sistemas operacionais que não são da Microsoft, como

Mac OS X e diversas distribuições Unix/Linux. Para montar o cenário, segue uma pequena lista de alguns dos recursos principais fornecidos como cortesia .NET:

- *Interoperabilidade completa com o código existente*: Isto é (obviamente) uma coisa boa. Os binários COM podem fundir-se (isto é, interoperar) com binários .NET mais novos e vice-versa. Além disso, o Platform-Invocation Services (PInvoke) permite que você chame bibliotecas baseadas em C (incluindo a API do sistema operacional) a partir do código .NET.
- *Integração de linguagem completa e total*: O .NET suporta herança, manipulação de exceções e depuração de código entre linguagens.
- *Um mecanismo de tempo de execução comum compartilhado por todas as linguagens compatíveis com .NET*: Um aspecto deste mecanismo é um conjunto bem definido de tipos que cada linguagem compatível com .NET "entende".
- *Uma biblioteca de classes básicas completa*: Esta biblioteca fornece proteção contra as complexidades dos chamados API puros e oferece um modelo de objeto consistente utilizado por todas as linguagens compatíveis com .NET entendem.
- *Não existe mais o encanamento COM*: `IClassFactory`, `IUnknown`, `IDispatch`, código IDL e os tipos de dados compatíveis com variáveis (BSTR, SAFEARRAY e assim por diante) não têm lugar em um binário .NET.
- *Um modelo de implementação realmente simplificado*: Com o .NET, não há necessidade de registrar uma unidade binária no registro do sistema. Além disso, o .NET permite que várias versões do mesmo `*.dll` existam em harmonia em uma única máquina.

Como pode concluir a partir dos pontos acima, a plataforma .NET não tem nada a ver com o COM (além do fato de que ambos os frameworks tenham origem na Microsoft). Na verdade, a única maneira de os tipos .NET e COM interagirem entre si é utilizando a camada de interoperabilidade.

■ **Nota** Uma abordagem da camada de interoperabilidade .NET pode ser vista no Apêndice A.

Apresentando os Blocos de Construção da Plataforma .NET (CLR, CTS e CLS)

Agora que conhece alguns dos benefícios fornecidos pelo .NET, falaremos sobre as três principais (e inter-relacionadas) entidades que tornam tudo isso possível: CLR, CTS e CLS. Do ponto de vista de um programador, o .NET pode ser compreendido como um ambiente de tempo de execução e uma biblioteca completa de classes básicas. A camada do tempo de execução é adequadamente citada como *common language runtime*, ou *CLR*. O papel principal do CLR é localizar, carregar e gerenciar tipos .NET em seu nome. O CLR também cuida de vários detalhes de nível mais baixo, como gerenciamento de memória; criação de domínios, threads e limites de contexto de objeto; e execução de diversas verificações de segurança.

Outro bloco de construção da plataforma .NET é o *Common Type System*, ou *CTS*. As especificações do CTS descrevem todos os tipos de dados e estruturas de programação possíveis suportados no tempo de execução, especificam como estas entidades podem interagir entre si e detalha como elas são representadas no formato de metadados .NET (você verá mais sobre metadados mais adiante neste capítulo; consulte o Capítulo 16 para detalhes completos).

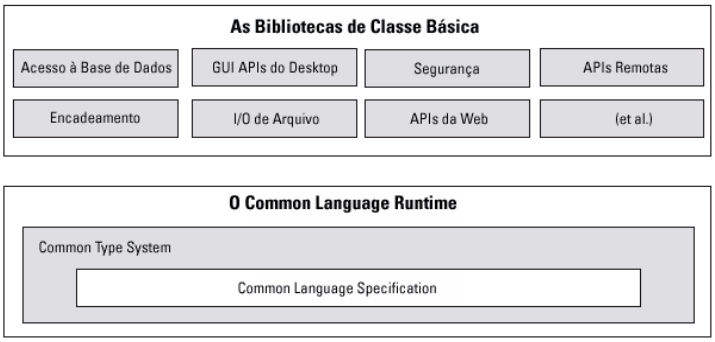
Compreenda que uma determinada linguagem compatível com .NET pode não suportar todos os recursos definidos pelo CTS. O *Common Language Specification (CLS)* é uma especificação relacionada, que define um subconjunto de tipos comuns e estruturas de programação com o qual todas as linguagens de programação .NET podem concordar. Portanto, se construir tipos .NET que só expõem recursos em conformidade com CLS, pode ter certeza de que todas as linguagens compatíveis com .NET poderão consumi-los. De modo contrário, se utilizar um tipo de dado ou uma estrutura de programação que esteja fora dos limites do CLS, não poderá garantir que todas as linguagens de programação .NET interajam

com sua biblioteca de código .NET.

O Papel das Bibliotecas de Classe Básica

Além das especificações CLR e CTS/CLS, a plataforma .NET fornece uma biblioteca de classe básica, que está disponível para todas as linguagens de programação .NET. esta biblioteca de classe não apenas encapsula diversas unidades básicas, como threads, entrada e saída de arquivos (I/O), renderização gráfica e interação com diversos dispositivos externos de hardware, como também fornece suporte a vários serviços exigidos pela maioria das aplicações do mundo real.

Por exemplo: as bibliotecas de classe básica definem tipos que facilitam o acesso à base de dados, manipulação de documentos XML, segurança de programação e construção de front-ends habilitados à web (bem como front-ends de desktop tradicionais e baseados em console. Em



um nível mais elevado, é possível visualizar a relação entre CLR, CTS, CLS e a biblioteca de classe básica, como mostra a Figura 1-1.

Figura 1-1. *Relação entre CLR, CTS, CLS e a biblioteca de classe básica*

A Contribuição do C#

Como o .NET é um desvio radical das tecnologias anteriores, a Microsoft criou uma nova linguagem de programação, C# (pronuncia-se "C sharp"), especificamente para esta nova plataforma. O C# é uma linguagem de programação cuja sintaxe principal é *bastante* similar à sintaxe Java. Entretanto, não é preciso chamar o C# de uma quebra do Java. Tanto C# quanto Java fazem parte da família C de linguagens de programação (C, Objective C, C++, etc.) e, portanto, compartilham uma sintaxe similar. Assim como o Java, de muitas maneiras, é uma versão mais limpa do C++, o C# pode ser visto como uma versão mais limpa do Java.

A verdade é que grande parte das estruturas sintáticas de C# é modelada de acordo com diversos aspectos do Visual Basic 6.0 e C++. Por exemplo: assim como o VB6, o C# suporta a noção das propriedades de tipo formal (ao contrário dos métodos tradicionais de getter e setter) e a capacidade de declarar métodos tomando um número variável de argumentos (através de arrays de parâmetros). Assim como C++, o C# permite que você sobrecarregue operadores e também crie estruturas, enumerações e funções de callback (através de delegates).

■ **Nota** Como verá no Capítulo 13, o C# 2008 adotou diversas estruturas encontradas tradicionalmente em diversas linguagens funcionais (por exemplo: LISP ou Haskell). Além disso, com a chegada do LINQ (consulte os Capítulos 14 e 24), o C# suporta diversas estruturasde programação,. O que o torna praticamente exclusivo no cenário da programação. Entretanto, o ponto principal do C# é a influência das linguagens baseadas em C.

Como o C# é um híbrido de diversas linguagens, o resultado é um produto que é sintaticamente limpo – se não mais limpo – que o Java, tão simples quanto o VB6 e fornece quase tanto poder e flexibilidade quanto o C++ (sem os bits feios). Eis uma lista parcial dos principais recursos C# encontrados em todas as versões da linguagem:

- Não são necessários operadores! Os programas C#, tipicamente, não necessitam

de manipulação direta de operadores (embora possa chegar a este nível se for absolutamente necessário).

- Gerenciamento automático de memória através do coletor de lixo (garbage collection). Sendo assim, C# não suporta a palavra-chave `delete`.
- Estruturas sintáticas formais para classes, interfaces, estruturas, enumerações e delegates.
- A capacidade similar à de C++ para sobrecarregar operadores para um tipo personalizado, sem a complexidade (por exemplo: garantir que "retornar `*this` para permitir encadeamento" não seja problema seu).
- Suporte à programação baseada em atributos. Este tipo de desenvolvimento permite que você anote tipos e seus membros para qualificar seu comportamento.

Com o lançamento do .NET 2.0 (por volta de 2005), a linguagem de programação C# foi atualizada para suportar diversos recursos novos, principalmente os seguintes:

- A capacidade de construir tipos genéricos e membros genéricos. Com eles, é possível construir códigomuito eficiente e tipado, que define diversos "alocadores de espaço" especificados no momento em que você interage com o item genérico.
- Suporte para métodos anônimos, que permite inserir uma função em linha onde um tipo delegate for necessário.
- Diversas simplificações para o modelo delegate/evento, inclusive co-variância, contra-variância e conversão do grupo de método.
- A capacidade de definir um único tipo em vários arquivos de código (ou, se necessário, como uma representação in-memory) utilizando a palavra-chave `partial`.

Como deve estar adivinhando, o .NET 3.5 confere ainda mais funcionalidade à linguagem de programação C# (C# 2008, para ser mais exato), inclusive os seguintes recursos:

- Suporte para queries fortemente tipadas (como LINQ, ou Language Integrated Query) utilizadas para interagir com diversas formas de dados
- Suporte a tipos anônimos, permitindo que você modele a "forma" de um tipo, e não o seu comportamento
- A capacidade de aumentar a funcionalidade de um tipo existente usando métodos de extensão
- Inclusão de um operador lambda `(=>)`,que simplifica ainda mais o trabalho com tipos delegate .NET
- Uma nova sintaxe de inicialização de objeto, que permite configurar valores de propriedade no momento da criação do objeto

Talvez o ponto mais importante a ser compreendido com relação à linguagem C# é que ela só pode produzir código executável dentro do runtime .NET (você nunca usaria o C# para construir um servidor COM nativo ou uma aplicação Win32 API não gerenciada). Oficialmente, o termo utilizado para descrever o código relacionado ao tempo de execução .NET é *código gerenciado*. A unidade binária que contém o código gerenciado é chamada de *assembly* (mais detalhes sobre assemblies logo adiante, na seção "Uma Visão Geral dos Assemblies .NET"). De modo contrário, um código que não pode ser hospedado diretamente pelo runtime .NET é chamado de *código não gerenciado*.

Outras Linguagens de Programação Compatíveis com .NET

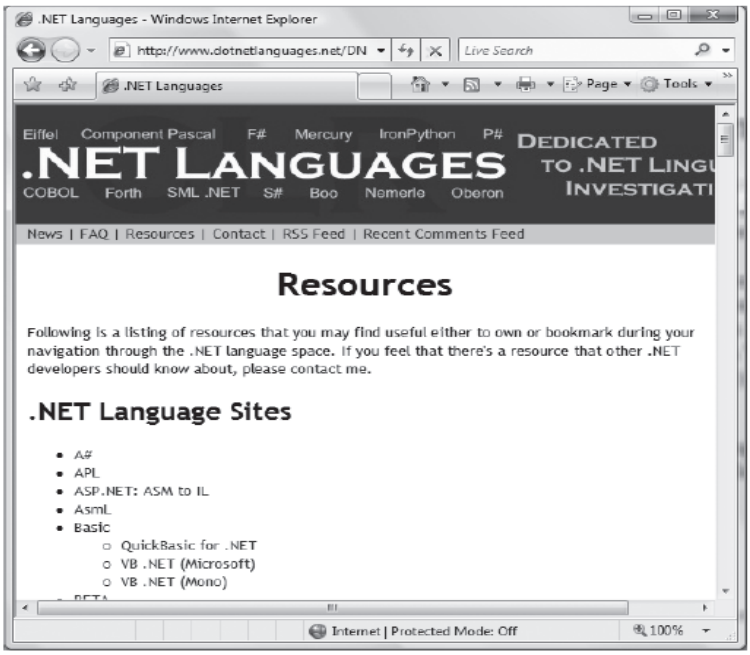
Entenda que o C# não é a única linguagem que pode ser usada para construir aplicações .NET. Quando a plataforma .NET foi apresentada pela primeira vez ao público, durante

a Microsoft Professional Developers Conference (PDC) de 2000, diversos fornecedores anunciaram que estavam muito ocupados construindo versões compatíveis com .NET de seus respectivos compiladores.

Desde então, dezenas de linguagens diferentes passaram por uma atualização para .NET. Além das cinco linguagens que vêm com o Microsoft .NET Framework 3.5 SDK (C#, Visual Basic .NET, J#, C++/CLI [antes chamada de "Extensões Gerenciadas para C++"] e JScript .NET), existem compiladores .NET para Smalltalk, COBOL e Pascal (para citar algumas). Embora este livro se concentre (quase) exclusivamente em C#, esteja a par do seguinte site web (por favor, saiba que esta RL está sujeita a mudanças):

```
http://www.dotnetlanguages.net
```

Se você clicar no link Resources no topo da página, encontrará uma lista de diversas linguagens de programação .NET e links relacionados onde pode baixar diversos compiladores (veja a Figura 1-2).



Ao mesmo tempo em que suponho que esteja interessado principalmente em construir programas .NET utilizando a sintaxe do C#, eu o incentivo a visitar este site, já que poderá encontrar muitas linguagens .NET que valem a pena ser investigadas por lazer (LISP .NET, alguém?).

Figura 1-2. *www.DotNetLanguages.net é um dos muitos sites que documentam linguagens de programação .NET conhecidas .*

A Vida em um Mundo Multilíngüe

Na primeira vez em que os desenvolvedores compreenderam a natureza de ignorância da linguagem .NET, diversas questões surgiram. A principal era: "Se todas as linguagens .NET são compiladas em 'código gerenciado', por que precisamos de mais de um compilador?" Existem diversas maneiras de responder a esta pergunta. Primeiro, nós, programadores, somos muito específicos quando se trata de escolher uma linguagem de programação (inclusive eu). Alguns de nós preferimos linguagens cheias de ponto-e-vírgula e chaves, com o menor número possível de palavras reservadas. Outros apreciam uma linguagem que ofereça indicações sintáticas mais "legíveis para os humanos" (como o Visual Basic). Outros ainda podem querer alavancar suas habilidades em mainframe ao mesmo tempo em que muda para a plataforma .NET (via COBOL .NET).

Agora, seja honesto. Se a Microsoft fosse desenvolver uma única linguagem .NET "official" derivada da família BASIC de linguagens, você diria que todos os programadores ficariam

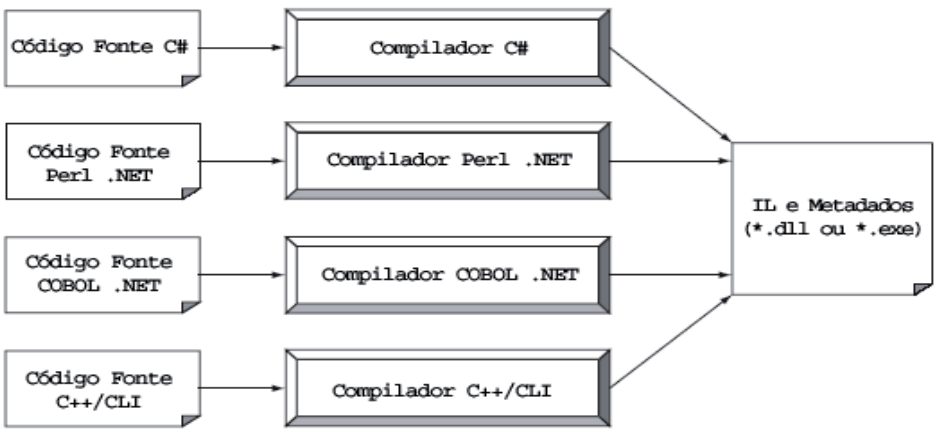
felizes com esta opção? Ou, se a linguagem .NET "official" fosse baseada na sintaxe Fortran, imagine quantos caras iriam ignorar o .NET como um todo. Como o tempo de execução .NET não se importa nem um pouco com a linguagem que foi utilizada para construir um bloco de código gerenciado, os programadores .NET podem permanecer fiéis às suas opções sintáticas e compartilhar os assemblies compilados com colegas de equipe, departamentos e organizações externas (não importa qual linguagem .NET os outros escolham usar).

Outro subproduto excelente da integração de diversas linguagens .NET em um software unificado é o simples fato de que todas as linguagens de programação têm seus próprios conjuntos de pontos fortes e fracos. Por exemplo: algumas linguagens de programação oferecem excelente suporte intrínseco para processamento matemático avançado. Outras oferecem suporte superior para cálculos financeiros, cálculos lógicos, interação com os computadores mainframe e assim por diante. Quando você pega os pontos fortes de uma determinada linguagem de programação e incorpora os benefícios fornecidos pela plataforma .NET, todos saem ganhando.

É claro que, na verdade, há chances muito boas de passar grande parte do seu tempo construindo software utilizando a linguagem .NET que escolher. Entretanto, quando aperfeiçoa a sintaxe de uma linguagem .NET, é muito fácil aprender outra. Isto também é bastante benéfico principalmente para os consultores de software do mundo todo. Se a linguagem que escolheu calha de ser C#, mas você está em um cliente que se comprometeu com Visual Basic .NET, ainda é possível alavancar a funcionalidade do .NET Framework e você deve conseguir compreender a estrutura geral da base de código com um mínimo de bagunça e preocupação. Já falei o suficiente.

Uma Visão Geral dos assemblies .NET

Não importa qual linguagem .NET você escolhe para programar, entenda que apesar de os binários .NET levarem a mesma extensão de arquivo que os servidores COM e binários Win32 não gerenciados (`*.dll` ou `*.exe`), eles não têm qualquer similaridade interna. Por exemplo, os binários .NET `*.dll` não exportam métodos para facilitar a comunicação com o tempo de execução COM (dado que .NET *não é* COM). Além disso, os binários .NET não são descritos utilizando as bibliotecas de tipo COM e não são registrados no registro do sistemas. O mais importante, talvez, é



que os binários .NET não contêm instruções específicas de uma plataforma, e sim uma linguagem intermediária (*IL, intermediate language*) que ignora a plataforma e metadados de tipos. A Figura 1-3 mostra a visão geral da história até agora.

Figura 1-3. *Todos os compiladores compatíveis com .NET emitem instruções IL e metadados.*

■ **Nota** Há um comentário a ser feito com relação à abreviação "IL". Durante o desenvolvimento do .NET, o termo oficial para a IL era Microsoft Intermediate Language (MSIL). Entretanto, com a versão final do .NET, o termo foi modificado para common intermediate language (CIL). Portanto, quando ler sobre .NET, entenda que IL, MSIL e CIL estão descrevendo exatamente a mesma entidade. Para manter a terminologia

atual, utilizarei a abreviação "CIL" neste texto.

Quando um `*.dll` ou um `*.exe` é criado com um compilador compatível com .NET, o módulo resultante é agrupado em um *assembly*. Irá examinar diversos detalhes dos assemblies .NET no Capítulo 15. Entretanto, para facilitar a discussão sobre o ambiente de tempo de execução .NET, precisa compreender algumas propriedades básicas deste novo formato de arquivo.

Como já foi mencionado, um assembly contém código CIL, que é conceitualmente familiar para o bytecode do Java, pois não é compilado para instruções específicas de uma plataforma até que seja absolutamente necessário. Tipicamente, "absolutamente necessário" é o ponto no qual um bloco de instruções CIL (como a implementação de um método) é referenciado para uso pelo tempo de execução .NET.

Além das instruções CIL, os assemblies também contêm *metadados* que descrevem em detalhes vivos as características de cada "tipo" existente no binário. Por exemplo: se você tem uma classe chamada SportsCar, os metadados de tipo descrevem os detalhes, como a classe básica de `SportsCar`, cujas interfaces são implementadas por `SportsCar` (se houver), bem como uma descrição completa de cada membro suportado pelo tipo `SportsCar`.

Os metadados .NET são uma melhora dramática para os metadados COM. Como já deve saber, os binários COM são tipicamente descritos através de uma biblioteca de tipos associados (que é pouco mais que uma versão binária do código Interface Definition Language [IDL]). Os problemas com as informações do tipo COM são que não há como garantir que eles estarão presentes, bem como o fato de que o código IDL não tem como documentar os servidores referenciados externamente e que são necessários para a operação correta do atual servidor COM. Em contraste, os metadados .NET estão sempre presentes e são gerados automaticamente por um determinado compilador compatível com .NET.

Finalmente, além do CIL e dos metadados de tipos, os próprios assemblies também são descritos utilizando metadados, o que é oficialmente chamado de *manifesto*. O manifesto contém informações sobre a versão atual do assembly, informações de cultura (utilizadas para localizar string e recursos de imagem) e uma lista de todos os assemblies referenciados externamente necessários para uma execução adequada. Você conhecerá várias ferramentas que podem ser utilizadas para examinar os tipos, metadados e informações de manifesto de um assembly ao longo dos próximos capítulos.

Assemblies Multifile e Não-Multifile

Na maioria dos casos, existe uma correspondência simples entre um assembly .NET e o arquivo binário (`*.dll ou *.exe`). Portanto, se você está construindo um `*.dll .NET`, é seguro considerar que o binário e o assembly sejam a mesma coisa. Da mesma forma, se está construindo uma aplicação desktop executável, o *.exe pode ser referenciado simplesmente como o próprio assembly. Como verá no Capítulo 15, entretanto, isto não é totalmente preciso. Tecnicamente, se um assembly é composto por um único módulo `*.dll` ou `*.exe`, você tem um *assembly não-multifile*. Os assemblies não-multifile contêm todos os CIL, metadados e manifestos associados, necessários em um pacote autônomo, único e bem definido.

Por outro lado, os *assemblies multifile* são compostos por diversos binários .NET, sendo que cada um é chamado de *módulo*. Ao construir um assembly multifile, um destes módulos (chamado de *modulo primário*) deve conter o manifesto do assembly (e, possivelmente, instruções e metadados para vários tipos). Os outros módulos relacionados contêm um manifesto no nível do módulo, CIL e metadados de tipo.

Como deve suspeitar, o modulo primário documenta o conjunto de módulos secundários necessários dentro do manifesto do assembly. Portanto, por que optaria por criar um assembly multifile? Quando particiona um assembly em módulos discretos, obtém uma opção de implementação mais flexível. Por exemplo: se um usuário está referenciando um assembly remoto que precisa ser baixado para sua máquina, o runtime só irá baixar os módulos necessários. Portanto, você é livre para conceituar seu assembly de maneira que os tipos requisitados com menos freqüência (como uma classe chamada `HardDriveReformatter`) sejam mantidos em um módulo stand-alone separado.

Em contraste, se todos os seus tipos estivessem em um assembly não-multifile, o usuário final poderia acabar baixando uma grande quantidade de dados que não são realmente

necessários (o que é, obviamente, uma perda de tempo). Então, como pode ver, um assembly é, na verdade, um *agrupamento lógico* de um ou mais módulos relacionados que têm intenção de serem inicialmente implementados e declarados como uma unidade.

O Papel da Common Intermediate Language

Vamos examinar o código CIL, os metadados de tipo e o manifesto do assembly com mais detalhes. CIL é uma linguagem que está acima de qualquer conjunto de instruções específicas de uma plataforma. Por exemplo: o código C# a seguir exemplifica uma calculadora comum. Não se preocupe com a sintaxe exata por enquanto, mas note o formato do método `Add()` na classe Calc:

```
// Calc.cs
using System;

namespace CalculatorExample
{
 // Esta classe contém o ponto de entrada da aplicação.
 class Program
 {
  static void Main()
    {
      Calc c = new Calc();
      int ans = c.Add(10, 84);
      Console.WriteLine("10 + 84 is {0}.", ans);
      // Espere que o usuário pressione a tecla Enter antes de desligar.
      Console.ReadLine();
    }
  }

  // A calculadora C#.
  class Calc
  {
     public int Add(int x, int y)
     { return x + y; }
  }
}
```

Ao compilar este arquivo de código utilizando o compilador C# (csc.exe), você obtém um assembly não-multifile `*.exe` que contém um manifesto, instruções CIL e metadados descrevendo cada aspecto das classes Calc e Program.

Nota O Capítulo 2 analisa os detalhes do código compilado utilizando o compilador C#, bem como a utilização dos IDE gráficos, como Visual Studio, Visual C# Express e SharpDevelop.

Por exemplo: se precisasse abrir este assembly utilizando `ildasm.exe` (que será analisado mais adiante neste capítulo), descobriria que o método `Add()` é representado através da CIL da seguinte maneira:

```
.method public hidebysig instance int32 Add(int32 x,
  int32 y) cil managed
{
  // Tamanho do código 9 (0x9)
  .maxstack 2
  .locals init (int32 V_0)
  IL_0000: nop
  IL_0001: ldarg.1
```

```
  IL_0002: ldarg.2
  IL_0003: add
  IL_0004: stloc.0
  IL_0005: br.s IL_0007
  IL_0007: ldloc.0
  IL_0008: ret
} // fim do método Calc::Add
```

Não se preocupe caso não consiga compreender a CIL resultante deste método – o Capítulo 19 falará sobre os fundamentos da linguagem de programação CIL. Concentre-se no fato de que o compilador C# emite CIL e não instruções específicas da plataforma.

Agora, lembre-se de que isso vale para todos os compiladores compatíveis com .NET. Para ilustrar, suponha que tenha criado a mesma aplicação usando Visual Basic .NET ao invés de C#:

```
' Calc.vb
Imports System

Namespace CalculatorExample
  ' Um "Módulo" VB está em uma classe que contém apenas
  ' membros estáticos.
  Module Program
    Sub Main()
      Dim c As New Calc
      Dim ans As Integer = c.Add(10, 84)
      Console.WriteLine(''10 + 84 is {0}.'', ans)
      Console.ReadLine()
    End Sub
  End Module

  Class Calc
   Public Function Add(ByVal x As Integer, ByVal y As Integer) As Integer
     Return x + y
   End Function
  End Class
End Namespace
```

Se você analisar a CIL para o método `Add()`, encontrará instruções similares (levemente ajustada pelo compilador VB .NET, `vbc.exe`):

```
.method public instance int32 Add(int32 x,
  int32 y) cil managed
{
  // Tamanho do código 8 (0x8)
  .maxstack 2
  .locals init (int32 V_0)
  IL_0000: ldarg.1
  IL_0001: ldarg.2
  IL_0002: add.ovf
  IL_0003: stloc.0
  IL_0004: br.s IL_0006
  IL_0006: ldloc.0
  IL_0007: ret
} // fim do método Calc::Add
```

■ Código Fonte Os arquivos de código `Calc.cs` e `Calc.vb` são incluídos no subdiretório Capítulo 1.

Benefícios da CIL

Neste ponto, deve estar imaginando exatamente o que se ganha compilando o código fonte em CIL ao invés de diretamente em um conjunto específico de instruções. Um benefício é a integração da linguagem. Como já viu, cada compilador compatível com .NET produz instruções CIL quase idênticas. Portanto, todas as linguagens são capazes de interagir em um cenário binário bem definido.

Além disso, como a CIL ignora a plataforma, o próprio .NET Framework também ignora a plataforma, fornecendo os mesmos benefícios aos quais os desenvolvedores de Java estão acostumados (isto é: uma única base de código rodando em diversos sistemas operacionais). Na verdade, existe um padrão internacional para a linguagem C# e um grande subconjunto da plataforma .NET e implementações já existe para muitos sistemas operacionais não-Windows (mais detalhes na seção "A Natureza Independente de Plataforma do .NET", mais adiante neste capítulo). Em contraste com Java, entretanto, o .NET permite que construa aplicações usando a linguagem de sua preferência.

Compilando CIL para Instruções Específicas de Plataforma

Como os assemblies contêm instruções CIL e não instruções específicas de plataforma, o código CIL deve ser compilado apressadamente antes do uso. A entidade que compila o código CIL em instruções que façam sentido para a CPU é chamada de *compilador just-in-time (JIT)*, que às vezes, é chamado pelo apelido carinhoso de *Jitter*. O ambiente de tempo de execução .NET cria um compilador IT para cada CPU voltada para o runtime, cada uma otimizada para a plataforma correspondente.

Por exemplo: se está construindo uma aplicação .NET que deve ser implementada em um dispositivo manipulado (como um Pocket PC), o Jitter correspondente está bem equipado para rodar em um ambiente com pouca memória. Por outro lado, se está implementando seu assembly em um servidor back-end (onde a memória raramente é um problema), o Jitter será otimizado para funcionar em um ambiente com muita memória. Desta forma, os desenvolvedores podem escrever um único corpo de código que possa ser compilado de forma eficiente pelo Jitter e executado em máquinas com diferentes arquiteturas.

Além disso, enquanto um determinado Jitter compila instruções CIL para o código da máquina correspondente, ele armazena os resultados em cachê na memória de modo a atender o sistema operacional de destino. Desta forma, se um chamado é feito para um método chamado `PrintDocument()`, as instruções CIL são compiladas em instruções de plataforma específica na primeira chamada e retidas na memória para uso futuro. Portanto, na próxima vez em que `PrintDocument()` for chamado, não haverá necessidade de recompilar a CIL.

■ **Nota** Também é possível fazer um "pré-JIT" de um assembly no momento da instalação da aplicação utilizando a ferramenta de linha de comando `ngen.exe`, que faz parte do .NET Framework 3.5 SDK. Esta ação pode melhorar o tempo de inicialização de aplicações com muitos gráficos.

O Papel dos Metadados de Tipos .NET

Além das instruções CIL, um assembly .NET contém metadados cheios, completos e precisos, que descrevem cada tipo (classe, estrutura, enumeração e assim por diante) definido no binário, bem como os membros de cada tipo (propriedades, métodos, eventos e assim por diante). Felizmente, é sempre trabalho do compilador (não do programador) emitir os metadados de tipos melhores e mais atualizados. Como os metadados .NET são tão cruelmente meticulosos, os assemblies são entidades completamente auto-descritas.

Para ilustrar o formato dos metadados de tipos .NET, daremos uma olhada nos metadados que foram gerados para o método `Add()` da classe C# Calc analisada anteriormente (os metadados gerados para a versão VB .NET do método `Add()` são similares):

```
TypeDef #2 (02000003)
-------------------------------------------------------
  TypDefName: CalculatorExample.Calc (02000003)
  Flags : [NotPublic] [AutoLayout] [Class]
  [AnsiClass] [BeforeFieldInit] (00100001)
```

```
  Extends : 01000001 [TypeRef] System.Object
  Method #1 (06000003)
-------------------------------------------------------
  MethodName: Add (06000003)
  Flags : [Public] [HideBySig] [ReuseSlot] (00000086)
  RVA : 0x00002090
  ImplFlags : [IL] [Managed] (00000000)
  CallCnvntn: [DEFAULT]
  hasThis
  ReturnType: I4
    2 Argumentos
    Argument #1: I4
    Argument #2: I4
    2 Parametros
    (1) ParamToken : (08000001) Name : x flags: [none] (00000000)
    (2) ParamToken : (08000002) Name : y flags: [none] (00000000)
```

Os metadados são utilizados por diversos aspectos do ambiente de tempo de execução .NET, bem como por diversas ferramentas de desenvolvimento. Por exemplo: o recurso IntelliSense oferecido por ferramentas como Visual Studio 2008 só é possível pela leitura dos metadados do assembly no tempo de desenvolvimento. Os metadados também são utilizados por diversos utilitários para navegação de objetos, ferramentas de depuração e o próprio compilador C#. Para ser mais preciso, os metadados são a espinha dorsal de diversas tecnologias .NET, inclusive Windows Communication Foundation (WCF), serviços web XML /a camada .NET remota, reflexos, ligação tardia e serialização de objetos. O Capítulo 16 irá formalizar o papel dos metadados .NET.

O Papel do Manifesto do Assembly

Por último, porém, não menos importante, lembre que um assembly .NET também contém metadados que descrevem o próprio assembly (tecnicamente, isto se chama *manifesto*). Entre outros detalhes, o manifesto documenta todos os assemblies externos necessários para que o assembly atual funcione corretamente, o número da versão do assembly, informações sobre direito de cópia e assim por diante. Assim como acontece com os metadados de tipo, é sempre trabalho do compilador gerar o manifesto do assembly. Eis alguns detalhes relevantes do manifesto gerado ao se compilar o arquivo de código `Calc.cs` mostrado anteriormente neste capítulo (suponha que tenhamos instruído o compilador a chamar seu assembly de `Calc.exe`):

```
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
  .ver 2:0:0:0
}
.assembly Calc
{
  .hash algorithm 0x00008004
  .ver 0:0:0:0
}
.module Calc.exe
.imagebase 0x00400000
.subsystem 0x00000003
.file alignment 512
.corflags 0x00000001
```

Resumindo, o manifesto documenta a lista de assemblies externos requisitados por `Calc.exe` (através da diretiva `.assembly extern`), bem como diversas características do assembly (número da versão, nome do módulo e assim por diante). O Capítulo 15 analisará a utilidade dos dados do manifesto com mais detalhes.

Compreendendo o Common Type System

Um determinado assembly pode conter qualquer número de tipos distintos. No mundo do .NET, *tipo* é simplesmente um termo geral usado para citar um membro do conjunto {classe,

interface, estrutura, enumeração, delegate}. Ao criar soluções utilizando uma linguagem compatível com .NET, provavelmente, você interage com muitos deste tipos. Por exemplo: seu assembly pode definir uma única classe que implementa um determinado número de interfaces. Talvez um dos métodos da interface interprete um tipo de enumeração como um parâmetro de entrada e retorne uma estrutura para o chamador.

Lembre que o CTS é uma especificação formal, que documenta a maneira como os tipos devem ser definidos para serem hospedados pelo CLR. Tipicamente, os únicos indivíduos que estão profundamente preocupados com o funcionamento interno do CTS são aqueles que desenvolvem ferramentas e/ou compiladores voltados à plataforma .NET. Entretanto, é importante que todos os programadores .NET aprendam como trabalhar com os cinco tipos definidos pelo CTS na linguagem escolhida. A seguir, uma breve visão geral.

Tipos de Classe CTS

Cada linguagem compatível com .NET suporta, no mínimo, a noção de um *tipo de classe*, que é a base da programação orientada a objetos (sigla em inglês: OOP [Object-Oriented Programming]). Uma classe pode ser composta por qualquer número de membros (como propriedades, métodos e eventos) e pontos de dados (campos). Em C#, as classes são declaradas através da palavra-chave `class`:

```
// Um tipo de classe C#.
class Calc
{
  public int Add(int x, int y)
  { return x + y; }
}
```

O Capítulo 5 iniciará sua análise de construção de tipos de classe CTS com C#; porém, a Tabela 1-1 documenta diversas características pertencentes aos tipos de classe.

Tabela 1-1. *Características da Classe CTS*

Característica da Classe	Função
A classe é "selada" ou não?	Classes seladas não podem funcionar como classe básica para outras classes.
A classe implementa alguma interface?	Uma *interface* é uma coleção de membros abstratos que fornece um contrato entre o objeto e seu usuário. O TS permite que uma classe ou estrutura implemente qualquer número de interfaces.
A classe é abstrata ou concreta?	Classes *abstratas* não podem ser criadas diretamente, mas têm intenção de definir comportamentos comuns para os tipos derivados. Classes *concretas* podem ser criadas diretamente.
Qual é a visibilidade desta classe?	Cada classe deve ser configurada com um atributo de visibilidade. Basicamente, este traço define se a classe pode ser usada por assemblies externos ou somente a partir do assembly destacado.

Tipos de Interface CTS

As *interfaces* não passam de uma coleção nomeada de definições abstratas de membro, que podem ser suportadas (isto é, implementadas) por uma determinada classe ou estrutura. Ao contrário das interfaces COM, as interfaces .NET *não* derivam uma interface de base comum, como IUnknown. Em C#, os tipos de interface são definidos através da palavra-chave interface, por exemplo:

```
// Um tipo de interface C# geralmente é
// declarado como público, para permitir que tipos de outros
```

```
// assemblies implementem seu comportamento.
public interface IDraw
{
  void Draw();
}
```

Sozinhas, as interfaces têm pouca utilidade. Entretanto, quando uma classe ou estrutura implementa uma determinada interface de maneira única, você pode requisitar acesso à funcionalidade fornecida utilizando uma referência à interface de maneira polimórfica. A programação baseada em interface será totalmente explorada no Capítulo 9.

Tipos de Estrutura CTS

O conceito de uma estrutura também é formalizado sob o CTS. Se você tem conhecimento de C, deve estar gostando de saber que estes tipos definidos pelo usuário sobreviveram no mundo .NET (embora eles se comportem de maneira um pouco diferente). Colocando de forma simples, uma *estrutura* pode ser vista como um tipo de classe leve, com semântica baseada em valores. Para obter mais detalhes sobre as sutilezas das estruturas, consulte o Capítulo 4. Tipicamente, as estruturas são mais adaptadas para modelar dados geométricos e matemáticos, e são criadas em C# através da palavra-chave `struct`:

```
// Um tipo de estrutura C#.
struct Point
{
  // As estruturas podem conter campos.
  public int xPos, yPos;

  // As estruturas podem conter construtores parametrizados.
  public Point(int x, int y)
  { xPos = x; yPos = y;}

  // As estruturas podem definir métodos.
  public void Display()
  {
    Console.WriteLine("({0}, {1}", xPos, yPos);
  }
}
```

Tipos de Enumeração CTS

Enumerações são estruturas de programação úteis, que permitem agrupar pares nome/valor. Por exemplo: suponha que esteja criando uma aplicação de vídeo-game que permita que o jogador selecione uma entre três categorias para o personagem (Assistente, Lutador ou Ladrão). Ao invés de rastrear valores numéricos puros para representar cada possibilidade, você poderia construir uma enumeração personalizada com a palavra-chave `enum`:

```
// Um tipo de enumeração C#.
enum CharacterType
{
  Wizard = 100,
  Fighter = 200,
  Thief = 300
}
```

Por padrão, o armazenamento utilizado para manter cada item é um inteiro de 32 bits; porém, também é possível alterar este slot de armazenamento se for necessário (por exemplo: ao programar para um dispositivo com pouca memória, como um Pocket PC). Além disso, o CTS exige que tipos enumerados derivem de uma classe básica comum, `System.Enum`. Como você verá no Capítulo 4, esta classe básica define vários membros interessantes, que permitem extrair, manipular e transformar automaticamente os pares nome/valor correspondentes.

Tipos de Delegate CTS

Delegates são o equivalente .NET de um ponteiro para função em estilo C tipada. A principal diferença é que um delegate .NET é uma *classe* derivada do `System.MulticastDelegate` e não um simples ponteiro para um endereço de memória puro. Em C#, os delegates são declarados através da plavra-chave `delegate`:

```
// Este tipo de delegate C# pode "apontar para" qualquer método
// retornando um inteiro e tomando dois inteiros como dado de entrada.
delegate int BinaryOp(int x, int y);
```

Os delegates são úteis quando você deseja fornecer um caminho para que uma entidade transmita uma chamada para outra entidade, bem como oferecer a base para a arquitetura de evento .NET. Como verá nos Capítulos 11 e 18, os delegates têm suporte intrínseco para multicasting (multidifusão) (isto é, transmitir uma solicitação para diversos recipientes) e invocações de método assíncrono.

Membros de Tipo CTS

Agora que visualizou cada um dos tipos formalizados pelo CTS, perceba que a maioria dos tipos aceita qualquer número de *membros*. Formalmente, um *membro de tipo* é restringido pelo conjunto {construtor, finalizador, construtor estático, tipo aninhado, operador, método, propriedade, indexador, campo, campo somente leitura, constante, evento}.

O CTS define diversos "adornos" que podem ser associados com um determinado membro. Por exemplo: cada membro tem um determinado traço de visibilidade (por exemplo: público, provado, protegido e assim por diante). Alguns membros podem ser declarados como abstratos, para forçar um comportamento polimórfico em tipos derivados, ou virtuais, para definir uma implementação encapsulada (mas substituível). Além disso, a maioria dos membros deve ser configurada como estática (ligada no nível da classe) ou como instância (ligada no nível do objeto). A construção de membros de tipo será analisada ao longo dos próximos capítulos.

■ **Nota** Como será descrito no Capítulo 10, a linguagem C# também suporta a construção de tipos e membros genéricos.

Tipos de Dados CTS Intrínsecos

O aspecto final do CTS de que se deve estar ciente por enquanto é que ele estabelece um conjunto bem definido de tipos de dados fundamentais. Embora uma determinada linguagem tipicamente tenha uma palavra-chave exclusiva, utilizada para declarar um tipo de dado CTS intrínseco, todas as palavras-chave da linguagem se reduzem ao mesmo tipo definido em um assembly chamado `mscorlib.dll`. Considere a Tabela 1-2, que documenta como os principais tipos de dados CTS são expressos em diversas linguagens .NET.

Tabela 1-2. *Os Tipos de Dados CTS Intrínsecos*

Tipo de Dado CTS	Palavra-chave VB .NET	Palavra-chave C#	Palavra-chave C++/CLI
`System.Byte`	`Byte`	`byte`	`unsigned char`
`System.SByte`	`SByte`	`sbyte`	`signed char`
`System.Int16`	`Short`	`short`	`short`
`System.Int32`	`Integer`	`int`	`int ou long`
`System.Int64`	`Long`	`long`	`_int64`
`System.UInt16`	`UShort`	`ushort`	`unsigned short`
`System.UInt32`	`UInteger`	`uint`	`unsigned int ou unsigned long`
`System.UInt64`	`ULong`	`ulong`	`unsigned __int64`
`System.Single`	`Single`	`float`	`Float`

System.Double	Double	Double	Double
System.Object	Object	object	Object^
System.Char	Char	char	wchar_t
System.String	String	string	String^
System.Decimal	Decimal	decimal	Decimal
System.Boolean	Boolean	bool	Bool

Como as palavras-chave exclusivas de uma linguagem gerenciada são apenas notações taquigrafadas para um tipo verdadeiro no namespace `System`, não temos mais que nos preocupar com condições de estouro/estouro negativo para dados numéricos ou como as strings e os Booleanos são representados internamente em diferentes linguagens. Considere os fragmentos de código a seguir, que definem variáveis numéricas de 32 bits em C# e VB .NET, utilizando palavras-chave da linguagem, bem como o tipo CTS formal:

```
// Definir alguns ''ints'' em C#.
int i = 0;
System.Int32 j = 0;

' Definir alguns ''ints'' em VB .NET.
Dim i As Integer = 0
Dim j As System.Int32 = 0
```

Compreendendo a Common Language Specification

Como deve estar ciente, diferentes linguagens expressam os mesmos constructs de programação em termos únicos, específicos à linguagem. Por exemplo: em C# você denota a concatenação de strings com o operador mais (+), enquanto que em VB.NET, tipicamente utiliza o ampersand (&). Mesmo quando duas linguagens diferentes expressam o mesmo dialeto programático (por exemplo: uma função sem valor de retorno), há chances muito grandes de que a sintaxe seja muito diferente na superfície:

```
// Método C# retornando nada.
public void MyMethod()
{
  // Um código muito interessante...
}
' Método VB retornando nada.
Public Sub MyMethod()
  ' Um código muito interessante...
End Sub
```

Como já viu, estas pequenas variações sintáticas são irrelevantes aos olhos do tempo de execução .NET, já que os respectivos compiladores (`csc.exe` ou `vbc.exe`, neste caso) emitem um conjunto similar de instruções CIL. Entretanto, as linguagens também podem ser diferentes com relação ao seu nível geral de funcionalidade. Por exemplo: uma linguagem .NET pode ou não ter uma palavra-chave para representar dados sem sinal e pode ou não suportar tipos de ponteiro. Dadas estas possíveis variações, seria ideal ter uma base com a qual todas as linguagens compatíveis com .NET devessem estar de acordo.

O CLS é um conjunto de regras que descreve em detalhes o pequeno, porém, completo conjunto de recursos que um determinado compilador compatível com .NET deve suportar para produzir código que possa ser hospedado pelo CLR e, ao mesmo tempo, possa ser acessado de maneira uniforme por todas as linguagens voltadas à plataforma .NET. De muitas maneiras, o CLS pode ser visto como um *subconjunto* da funcionalidade total definida pelo CTS.

Finalmente, o CLS é um conjunto de regras com as quais os construtores de compilador devem estar de acordo se quiserem que seus produtos funcionem perfeitamente no universo .NET. Cada regra recebe um nome simples (por exemplo "Regra 6 do CLS") e descreve como ela afeta aqueles que constroem os compiladores, bem como aquele que (de certa forma) interage

com eles. O melhor do CLS é a poderosa Regra 1:

- *Regra1*: As regras CLS aplicam-se apenas às partes de um tipo que são expostas fora do assembly principal.

Com esta regra, é possível inferir (corretamente) que o restante das regras do CLS não se aplica à lógica utilizada para construir as funções internas de um tipo .NET. Os únicos aspectos de um tipo que devem estar de acordo com o CLS são as próprias definições dos membros (isto é: convenções de nomes, parâmetros e tipos de retorno). A lógica de implementação para um membro pode utilizar qualquer número de técnicas que não sejam CLS, já que o mundo externo não saberá a diferença.

Para ilustrar, o método `Add()` a seguir não é compatível com CLS, já que os parâmetros e os valores de retorno utilizam dados sem sinal (que não é um requisito do CLS):

```
class Calc
{
  // Dados sem sinal expostos não são compatíveis com CLS!
  public ulong Add(ulong x, ulong y)
  { return x + y;}
}
```

Entretanto, se apenas utilizar os dados sem sinal internamente, como segue:

```
class Calc
{
  public int Add(int x, int y)
  {
    // Como esta variável ulong só é usada internamente,
    // ainda somos compatíveis com CLS.
    ulong temp = 0;
    ...
    return x + y;
  }
}
```

Você ainda estaria de acordo com as regras do CLS e poderia ter certeza de que todas as linguagens .NET seriam capazes de chamar o método `Add()`.

É claro que, além da Regra 1, o CLS define diversas outras regras. Por exemplo: o CLS descreve como uma determinada linguagem deve representar strings de texto, como as enumerações devem ser representadas internamente (o tipo básico usado para armazenamento), como definir membros estáticos e assim por diante.

Felizmente, você não tem que decorar estas regras para ser um desenvolvedor .NET proficiente. Novamente, um bom entendimento das especificações CTS e CLS é interessante apenas para os construtores de ferramentas/compiladores.

Garantindo a Compatibilidade com o CLS

Como verá ao longo deste livro, o C# define diversas estruturas de programação que não são compatíveis com CLS. A boa notícia, no entanto, é que você pode instruir o compilador C# a verificar seu código para compatibilidade com CLS utilizando um único atributo .NET:

```
// Pedir ao compilador C# que verifique a compatibilidade com CLS.
[assembly: System.CLSCompliant(true)]
```

O Capítulo 16 fornecerá mais detalhes da programação baseada em atributos. Até lá, apenas compreenda que o atributo `[CLSCompliant]` instruirá o compilador C# a verificar cada linha do código com relação às regras do CLS. Caso seja descoberta qualquer violação do CLS, você receberá um erro do compilador e uma descrição do código ofendido.

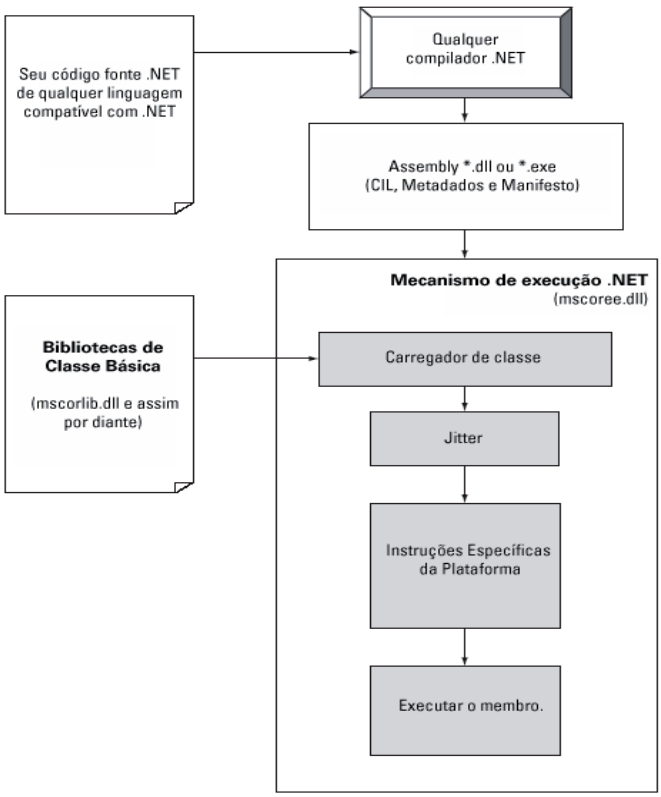
Compreendendo o Common Language Runtime

Além das especificações CTS e CLS, a última abreviação de três letras com a qual trabalharemos é o CLR. Em termos de programação, o termo *runtime* pode ser compreendido como uma coleção de serviços externos necessários para a execução de uma determinada unidade compilada de código. Por exemplo: quando os desenvolvedores utilizam o MCF para criar uma nova aplicação, eles estão cientes de que seu programa exige a biblioteca de runtime MFC (isto é, `mfc42.dll`). Outras linguagens populares também têm um runtime correspondente. Os programadores de VB6 também estão presos a um ou dois módulos de runtime (por exemplo: `msvbvm60.dll`). Os desenvolvedores de Java estão presos ao Java Virtual Machine (JVM), e assim por diante.

A plataforma .NET oferece mais um sistema de runtime. A principal diferença entre o runtime .NET e os outros runtimes que acabei de mencionar é o fato de que o runtime .NET fornece uma única e bem definida camada de runtime compartilhada por *todas* as linguagens e plataformas compatíveis com .NET.

O ponto central do CLR é representado fisicamente por uma biblioteca chamada `mscoree.dll` (mais conhecida como Common Object Runtime Execution Engine). Quando um assembly é referenciado para uso, a biblioteca `mscoree.dll` é carregada automaticamente e, por sua vez, carrega o assembly solicitado na memória. O mecanismo de runtime é responsável por diversas tarefas. Em primeiro lugar e mais importante, ele é a entidade encarregada de analisar o local de um assembly e encontrar o tipo solicitado dentro do binário através da leitura dos metadados. Em seguida, o CLR determina o tipo na memória, compila a CIL associada de acordo com as instruções específicas da plataforma, faz as verificações de segurança necessárias e, em seguida, executa o código em questão.

Além de carregar seus assemblies personalizados e de criar seus tipos personalizados, o CLR também interage com os tipos contidos nas bibliotecas de classe básica .NET quando necessário. Embora toda a biblioteca de classe básica tenha sido dividida em diversos assemblies discretos,



o assembly principal é o `mscoree.dll`. `mscoree.dll` contém um grande número de tipos centrais, que encapsulam uma grande variedade de tarefas comuns de programação, bem como

os tipos de dados centrais utilizados por todas as linguagens .NET. Ao construir soluções .NET, automaticamente tem acesso a este assembly em particular.

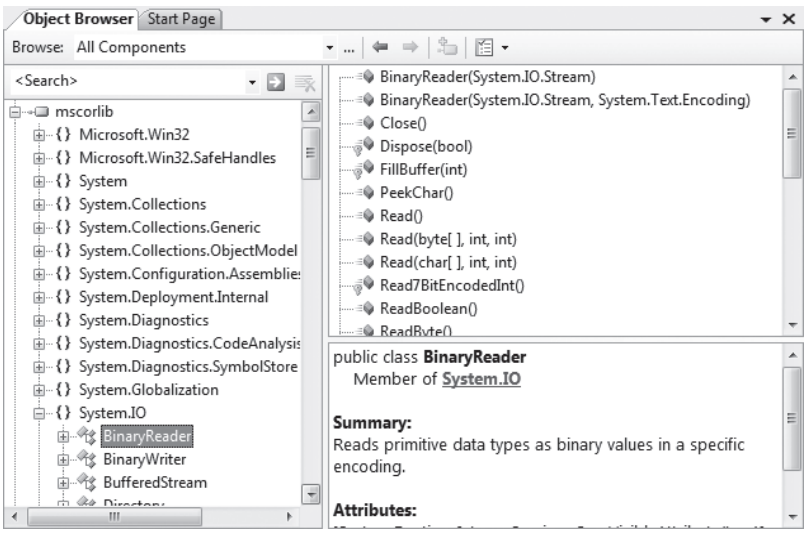
A Figura 1-4 ilustra o fluxo de trabalho que ocorre entre seu código fonte (que está usando os tipos da biblioteca de classe básica), um determinado compilador .NET e o mecanismo de execução .NET.

Figura 1-4. `mscoree.dll` *em ação*

A Distinção Assembly/Namespace/Tipo

Cada um de nós compreende a importância das bibliotecas de código. O objetivo de bibliotecas como MFC, J2EE e ATL é oferecer aos desenvolvedores um conjunto bem definido de código existente que possa ser usado em suas aplicações. Entretanto, a linguagem C# não vem com uma biblioteca de código específica para esta linguagem. Ao invés disso, os desenvolvedores de C# utilizam bibliotecas .NET de linguagem neutra. Para manter todos os tipos das bibliotecas de classe básica bem organizados, a plataforma .NET faz uso extensivo do conceito de *namespace*.

Colocando de forma simples, um namespace é um agrupamento de tipos relacionados semanticamente contidos em um assembly. Por exemplo: o namespace `System.IO` contém tipos



relacionados a arquivos I/O, o namespace `System.Data` define tipos básicos e assim por diante. É muito importante apontar que um único assembly (como `mscorlib.dll`) pode conter diversos namespaces, e cada um deles pode conter qualquer quantidade de tipos.

Para esclarecer, a Figura 1-5 mostra uma tela do Visual Studio 2008 Object Browser. Esta ferramenta permite que analise os assemblies referenciados pelo seu projeto atual, os namespaces de um assembly específico, os tipos contidos em um determinado namespace e os membros de um tipo específico. Note que `mscorlib.dll` contém muitos namespaces diferentes e cada um tem seus próprios tipos relacionados semanticamente.

Figura 1-5. *Um único assembly pode ter qualquer número de namespaces.*

A principal diferença entre esta abordagem e uma biblioteca específica de uma linguagem, como MFC, é que qualquer linguagem voltada para o runtime .NET utiliza os *mesmos* namespaces e os *mesmos* tipos. Por exemplo: os três programas a seguir ilustram a aplicação ubíqua "Hello World", escrita em C#, VB .NET e C++/CLI:

```
// Hello world em C#
using System;

public class MyApp
```

```
{
  static void Main()
  {
    Console.WriteLine(''Hi from C#'');
  }
}

' Hello world em VB
Imports System

Public Module MyApp
  Sub Main()
    Console.WriteLine(''Hi from VB'')
  End Sub
End Module

// Hello world in C++/CLI
#include ''stdafx.h''
using namespace System;

int main(array<System::String ^> ^args)
{
  Console::WriteLine(L'Hi from C++/CLI'');
  return 0;
}
```

Note que cada linguagem está utilizando a classe `Console` definida no namespace `System`. Além de pequenas variações sintáticas, estas três aplicações parecem muito similares, tanto física quanto logicamente.

Claramente, seu principal objetivo como desenvolvedor .NET é conhecer as riquezas de tipos definidos nos (diversos) namespaces .NET. O principal namespace no qual deve pôr as mãos é chamado `System`. Este namespace fornece um corpo de tipos dos quais precisará para ganhar tempo. Na verdade, você não poderá construir qualquer tipo de aplicação C# funcional sem pelo menos fazer uma referência ao namespace `System`, pois os tipos de dados centrais (`System.Int32`, `System.String` etc.) são definidos nele. A Tabela 1-3 oferece uma lista de alguns (mas certamente não de todos) os namespaces .NET agrupados por funcionalidade relativa.

Tabela1-3. *Uma Amostragem dos Namespaces .NET*

Namespace .NET	FUNÇÃO
`System`	Dentro de `System`, você encontra diversos tipos úteis, que lidam com dados intrínsecos, operações matemáticas, geração randômica de números, variáveis de ambiente e coletor de lixo, bem como diversas exceções e atributos usadas com freqüência.
`System.Collections` `Collections.Generic`	Estes namespaces definem diversos tipos de recipientes, bem `System.` como os tipos básicos e interfaces que permitem construir coleções personalizadas.

`System.Data`
`System.Data.Odbc`
`System.Data.OracleClient`
`System.Data.OleDb`
`System.Data.SqlClient`
	Estes namespaces são usadas para fazer a interface com as bases de dados relacionais utilizando ADO.NET.

`System.IO`
`System.IO.Compression`
`IO.Ports`
	Estes namespaces definem diversos tipos usados para trabalhar com entrada/saída de arquivos, compressão de dados e `System.` manipulação de portas.

`System.Reflection` `Reflection.Emit`	Estes namespaces definem tipos que suportam descoberta de `System.` tipo de runtime, bem como criação dinâmica de tipos.
`System.Runtime.Interop-` `Services`	Este namespace fornece facilidades para permitir que tipos .NET interajam com "código não gerenciado" (por exemplo: DLLs baseadas em C e servidores COM) e vice-versa.
`System.Drawing` `Windows.Forms`	Estes namespaces definem tipos utilizados para construir `System.` aplicações de desktop utilizando o kit de ferramentas de IU original do .NET (Windows Forms).

`System.Windows`
`Windows.Controls`
`System.Windows.Shapes`
	O namespace `System.Windows` é a raiz para diversos `System.` namespaces que representam o kit de ferramentas da IU do Windows Presentation Foundation (WPF).

`System.Linq`
`System.Xml.Linq`
`System.Data.Linq`
	Estes namespaces definem tipos usados na programação contra o LINQ API.

`System.Web`	Este é um dos diversos namespaces que permitem construir aplicações web ASP.NET.
`System.ServiceModel`	Este é um entre diversos namespaces utilizados para construir aplicações distribuídas utilizando o WCF API.
`System.Workflow.Runtime` `System.Workflow.Activities`	Estes são dois entre diversos namespaces que definem os tipos usados para construir aplicações "capacitadas para fluxo de trabalho" utilizando o WCF API.
`System.Threading`	Este namespace define diversos tipos para construir aplicações multitarefadas.
`System.Security`	Segurança é um aspecto integrado do universo .NET. Nos namespaces centrados em segurança, você encontra diversos tipos lidando com permissões, criptografia e assim por diante.
`System.Xml`	Os namespaces centrados em XML contêm diversos tipos utilizados para interagir com dados XML.

O Papel dos Namespaces Microsoft

Tenho certeza de que notou enquanto lia a listagem da Tabela 1-3 que `System` é o namespace raiz para um grande número de namespaces aninhados (`System.IO`, `System.Data` etc.). Como podemos perceber, entretanto, a biblioteca de classe básica .NET define diversos dos principais namespaces raiz além de `System`, dentre os quais, o mais útil chama-se `Microsoft`.

Resumindo, qualquer namespace aninhado dentro de `Microsoft` (por exemplo: `Microsoft.CSharp`, `Microsoft.Ink`, `Microsoft.ManagementConsole` e `Microsoft.Win32`) contém tipos utilizados para interagir com os serviços exclusivos do sistema operacional Windows. Tendo isto em mente, não suponha que estes tipos poderiam ser usados com sucesso em outros sistemas operacionais habilitados para .NET, como Mac OS X. Em sua grande maioria, este texto não irá aprofundar-se em detalhes dos namespaces enraizados em `Microsoft`, portanto, certifique-se de consultar a documentação se estiver muito interessado.

> **Nota** O Capítulo 2 irá ilustrar a utilização da documentação .NET Framework 3.5 SDK, que fornece detalhes sobr cada namespace, tipo e membro encontrado dentro das bibliotecas de classe básica.

Acessando um Namespace Programaticamente

Vale a pena reiterar que um namespace não passa de uma maneira conveniente para nós, meros humanos, compreendermos e organizarmos logicamente tipos relacionados. Considere novamente o namespace `System`. De sua perspectiva, pode supor que `System.Console` representa uma classe chamada `Console`, contida dentro de um namespace chamado `System`. Entretanto, aos olhos do runtime .NET, não é bem isso. O mecanismo do runtime só enxerga uma única entidade chamada `System.Console`.

Em C#, a palavra-chave `using` simplifica o processo de referenciar tipos definidos em um

determinado namespace. Eis como funciona: digamos que esteja interessado em construir uma aplicação de desktop tradicional. A janela principal renderiza um gráfico de barras com base em algumas informações obtidas em uma base de dados back-end e exibe o logotipo de sua empresa. Enquanto aprende os tipos contidos em cada namespace, ela estuda e experimenta. Estes são alguns possíveis candidatos para serem referenciados em seu programa:

```
// Estes são todos os namespaces usados para construir esta aplicação.
using System;                 // General base class library types.
using System.Drawing;       // Graphical rendering types.
using System.Windows.Forms; // Windows Forms GUI widget types.
using System.Data;            // General data-centric types.
using System.Data.SqlClient;  // MS SQL Server data access types.
```

Quando tiver especificado uma determinada quantidade de namespaces (e configurado uma referência para os assemblies que os definem), estará livre para criar instâncias para os tipos contidos neles. Por exemplo: se está interessado em criar uma instância para a classe `Bitmap` (definida no namespace `System.Drawing`), pode escrever:

```
// Listar explicitamente os namespaces usados por este arquivo.
using System;
using System.Drawing;

class Program
{
  public void DisplayLogo()
  {
    // Criar um bitmap de 20 por 20 pixels.
    Bitmap companyLogo = new Bitmap(20, 20);
    ...
  }
}
```

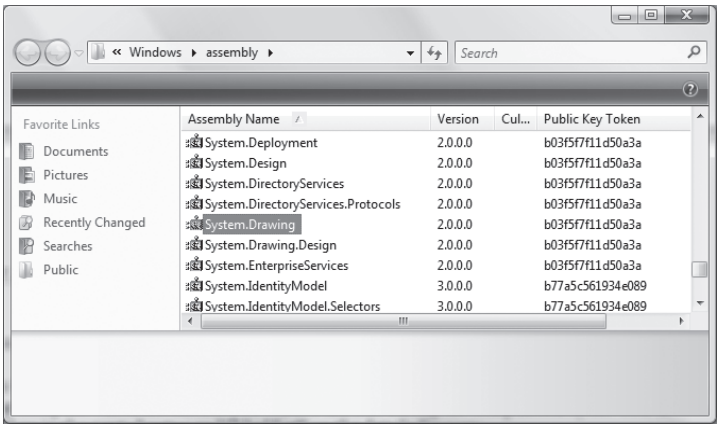
Como seu arquivo de código está referenciando `System.Drawing`, o compilador consegue visualizar a classe `Bitmap` como um membro deste namespace. Caso não tenha especificado o namespace `System.Drawing`, receberá um erro do compilador. No entanto, está livre para declarar variáveis utilizando um *nome totalmente qualificado*:

```
// Não está listando namespace System.Drawing!
using System;

class Program
{
  public void DisplayLogo()
  {
    // Utilizando nome totalmente qualificado.
    System.Drawing.Bitmap companyLogo =
      new System.Drawing.Bitmap(20, 20);
    ...
  }
}
```

Enquanto definir um tipo utilizando o nome totalmente qualificado oferece mais legibilidade, acho que concordaria que a palavra-chave `using` do C# reduz a digitação. Neste texto, evitarei a utilização de nomes totalmente qualificados (a não ser que haja uma ambigüidade definida a ser solucionada) e optarei pela abordagem simplificada da palavra-chave `using` do C#.

Entretanto, lembre-se sempre de que a palavra-chave `using` é apenas uma notação resumida para especificar o nome totalmente qualificado de um tipo, e qualquer uma das abordagens resulta *exatamente* na mesma CIL correspondente (dado o fato de que o código CIL sempre utiliza nomes totalmente qualificados) e não afeta o desempenho ou o tamanho do assembly.



Referenciando Assemblies Externos

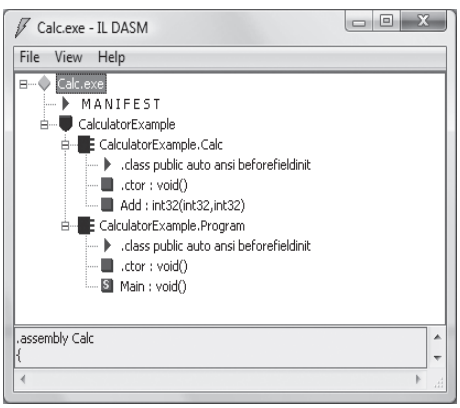
Além de especificar um namespace através da palavra-chave `using` do C#, você também precisa dizer ao compilador C# o nome do assembly que contém a definição CIL atualizada para o tipo referenciado. Como mencionado, muitos namespaces .NET centrais ficam dentro de `mscorlib.dll`. Entretanto, o tipo `System.Drawing.Bitmap` está contido em um assembly separado chamado `System.Drawing.dll`. Uma grande maioria dos assemblies do .NET Framework está localizada sob um diretório específico chamado de *global assembly cache* (GAC). Em uma máquina Windows, ele pode estar localizado sob C:\Windows\Assembly, como mostra a Figura 1-6.

Figura 1-6. *As bibliotecas de classe básica ficam no GAC.*

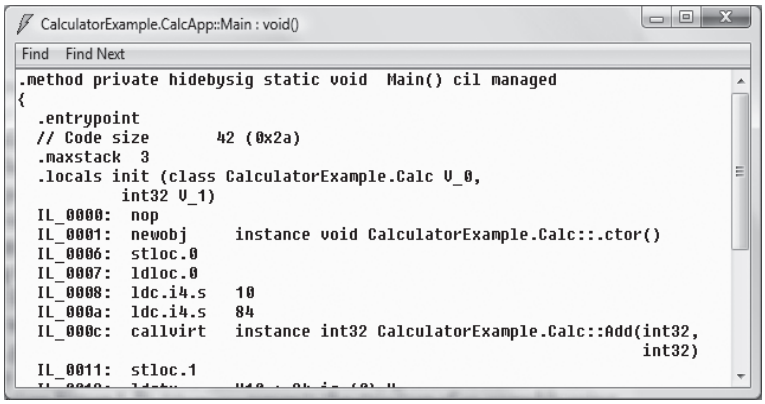
Dependendo da ferramenta de desenvolvimento que está usando para construir suas aplicações .NET, terá vários modos de informar ao compilador quais assemblies deseja incluir durante o ciclo de compilação. Você verá como fazer isto neste capítulo, portanto, não vou falar sobre os detalhes por enquanto.

Explorando um Assembly Usando o ildasm.exe

Se você está começando a se sentir um pouquinho surpreso com o pensamento de se aperfeiçoar em cada namespace da plataforma .NET, lembre que o que torna um namespace único é o fato dele conter tipos que estão, de alguma maneira, *semanticamente relacionados*. Portanto, se você não precisa de uma interface de usuário além de uma simples aplicação de console, pode esquecer tudo sobre os namespaces `System.Windows.Forms`, `System.Windows` e `System.Web` (entre outros). Se você está construindo uma aplicação de pintura, os namespaces da base de dados não merecem muita preocupação. Assim como o novo conjunto de código pré-fabricado, que aprenderá conforme usa.



O utilitário Intermediate Language Disassembler (`ildasm.exe`), que vem com o .NET Framework 3.5 SDK, permite que carregue qualquer assembly .NET e investigue seu conteúdo,

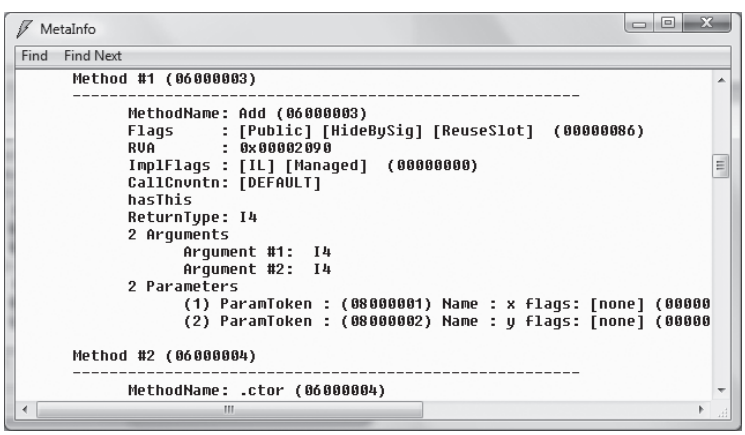


inclusive o manifesto associado, o código CIL e os metadados de tipo. Para carregar o `ildasm.exe`, abra um prompt de comando do Visual Studio (utilizando Iniciar ➤ Todos os Programas ➤ Microsoft Visual Studio 2008 ➤ Visual Studio Tools), digite `ildasm` e pressione a tecla Enter.

Quando a ferramenta abrir, acesse o menu de comandos Arquivo ➤ Abrir e navegue para um assembly que deseja explorar. Para efeito de ilustração, segue o assembly `Calc.exe` gerado com base no arquivo `Calc.cs` mostrado anteriormente neste capítulo (veja a Figura 1-7). O `ildasm.exe` apresenta a estrutura de um assembly usando um formato tree-view familiar.

Figura 1-7. *O* `ildasm.exe` *permite que visualize o código CIL, o manifesto e os metadados de um assembly .NET.*

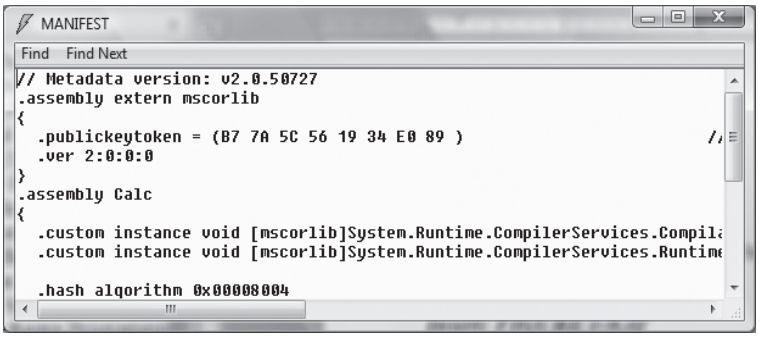
Figura 1-8. *Visualizando a CIL correspondente.*



Visualizando o Código CIL

Além de exibir os namespaces, tipos e membros contidos em um determinado assembly, o `ildasm.exe` também permite que visualize as instruções CIL para um determinado membro. Por exemplo: se fosse dar um clique duplo sobre o método `Main()` da classe `Program`, uma janela separada exibiria a CL correspondente (veja a Figura 1-8).

Visualizando Metadados de Tipo



Caso deseje visualizar os metadados de tipo para o assembly carregado atualmente, pressione Ctrl+M. A Figura 1-9 mostra os metadados para o método `Calc.Add()`.

Figura 1-9. *Visualizando metadados de tipo através do* `ildasm.exe`

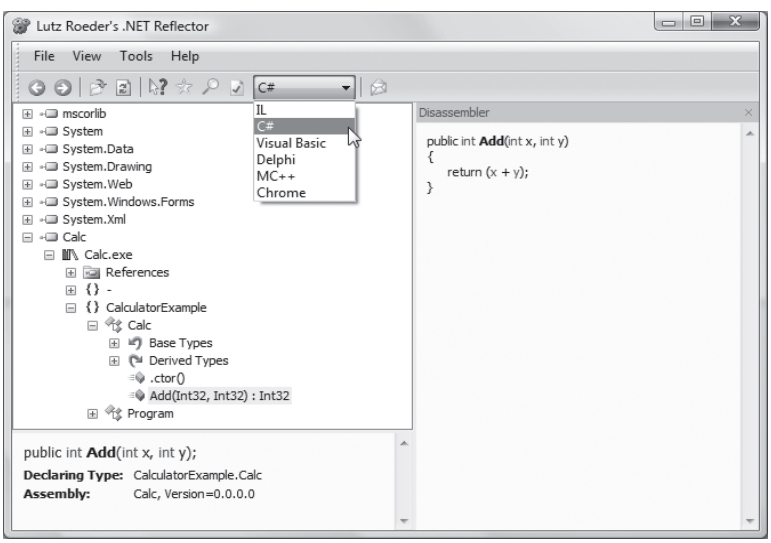
Visualizando Metadados de Assembly (mais conhecido como o Manifesto)

Finalmente, se está interessado em visualizar o conteúdo do manifesto do assembly, apenas dê um clique duplo sobre o ícone MANIFEST (veja a Figura 1-10).

Figura 1-10. *Visualizando os dados do manifesto através do* `ildasm.exe`.

Para sua informação, o `ildasm.exe` tem mais opções do que as exibidas aqui e ilustrarei os recursos adicionais da ferramenta quando for apropriado ao longo do texto.

Explorando um Assembly Utilizando o Reflector, de Lutz Roeder



Enquanto utilizar o `ildasm.exe` é uma tarefa muito comum quando deseja ir mais a fundo em um binário .NET, a única pegadinha é que você só é capaz de visualizar o código CIL correspondente, ao invés de olhar para a implementação de um assembly utilizando a linguagem gerenciada que escolheu. Felizmente, muitos navegadores de objeto.NET estão disponíveis para download, inclusive o popular Reflector.

Esta ferramenta gratuita pode ser baixada no endereço `http://www.aisto.com/roeder/dotnet`. Quando você descompacta o arquivo, pode rodar a ferramenta e conectar qualquer assembly que desejar utilizando a opção de menu Arquivo ➤ Abrir. A Figura 1-11 mostra nossa aplicação `Calc.exe` novamente.

Figura 1-11. *O Reflector é uma ferramenta de navegação de objetos muito popular.*

Note que `reflector.exe` suporta uma janela Disassembler (aberta quando se pressiona a barra de espaço) bem como uma caixa de lista drop-down, que permite visualizar a base de código correspondente na linguagem escolhida (incluindo, é claro, o código CIL).

Deixarei a seu critério a verificação do número de recursos intrigantes encontrados nesta ferramenta. Porém, esteja ciente de que, ao longo deste livro, utilizarei tanto o `ildasm.exe` quanto o `reflector.exe` para ilustrar diversos conceitos.

Implementando o Runtime .NET

Não deve ser surpresa que os assemblies .NET podem ser executados somente em uma máquina que tenha o .NET Framework instalado. Para um indivíduo que constrói software .NET, isso nunca deverá ser um problema, já que sua máquina de desenvolvimento estará configurada adequadamente no momento em que instalar o .NET Framework 3.5 SDK gratuito (bem como um ambiente de desenvolvimento .NET comercial, como o Visual Studio 2008).

Entretanto, se implementar um assembly em um computador que não tenha o .NET instalado, ele não conseguirá rodar. Por este motivo, a Microsoft fornece um pacote de configuração chamado `dotnetfx3setup.exe`, que pode ser enviado gratuitamente e instalado com seu software .NET. Este programa de instalação pode ser baixado gratuitamente na Microsoft, a partir de sua área de downloads (`http://msdn.microsoft.com/netframework`). Quando o `dotNetFx35setup.exe` estiver instalado, a máquina de destino conterá as bibliotecas de classe básica .NET, o runtime .NET (`mscoree.dll`) e a infra-estrutura .NET adicional (como o GAC).

■ **Nota** O sistema operacional Vista vem pré-configurado com a infra-estrutura de runtime .NET necessária. Entretanto, se está implementando sua aplicação em Windows XP ou em Windows Server 2003, desejará garantir que a máquina de destino tenha o ambiente de runtime .NET instalado e configurado.

A Natureza Independente de Plataforma do .NET

Para encerrar este capítulo, permita-me comentar brevemente sobre a natureza independente de plataforma da plataforma .NET. Para surpresa da maioria dos desenvolvedores, os assemblies .NET podem ser desenvolvidos e executados em sistemas operacionais não-Microsoft (Mac OS X, diversas versões do Linux e Solaris, para citar alguns). Para compreender como isto é possível, é preciso aceitar mais uma abreviação do universo .NET: CLI (Common Language Infrastructure).

Quando a Microsoft lançou a linguagem de programação C# e a plataforma .NET, ela também criou um conjunto de documentos formais que descreviam a sintaxe e a semântica do C# e das linguagens CIL, o formato do assembly .NET, os principais namespaces .NET e o mecanismo de um runtime .NET hipotético (conhecido como Virtual Execution System, ou VES).

Melhor ainda, estes documentos foram submetidos (e ratificados) à ECMA International para verificação de seus padrões internacionais (`http://www.ecma-international.org`). As especificações de interesse são:

- *ECMA-334*: Especificações da Linguagem C#
- *ECMA-335*: A Common Language Infrastructure (CLI)

A importância destes documentos fica clara quando você compreende que eles permitem que terceiros construam distribuições da plataforma .NET para qualquer quantidade de sistemas operacionais e/ou processadores. A ECMA-335 é, talvez, a mais "apetitosa" das duas especificações, tanto que foi dividida em várias partes, inclusive as mostradas na Tabela 1-4.

Tabela 1-4. *Divisões da CLI*

Divisões da ECMA-335	Função
Divisão I: Arquitetura	Descreve a arquitetura geral da CLI, inclusive as regras do CTS e CLS, e o funcionamento do mecanismo runtime .NET
Divisão II: Metadados	Descreve os detalhes dos metadados .NET
Divisão III: CIL	Descreve a sintaxe e a semântica do código CIL
Divisão IV: Bibliotecas	Fornece uma visão geral de alto nível das bibliotecas de classe mínimas e completas que devem ser suportadas por uma distribuição .NET
Divisão V: Anexos	Fornece uma coleção de detalhes gerais, como orientações para o projeto de uma biblioteca de classe e os detalhes de implementação de um compilador CIL

Veja que a Divisão IV (Bibliotecas) define somente um conjunto *mínimo* de namespaces que representam os serviços principais esperados por uma distribuição CLI (coleções, I/O de console, I/O de arquivo, encadeamento, reflexão, acesso a network, principais necessidades de segurança, manipulação de XML e assim por diante). A CLI *não* define namespaces que facilitam o desenvolvimento Web (ASP.NET), acesso à base de dados (ADO.NET), ou desenvolvimento de aplicação para interface gráfica de usuário para desktop (GUI) (Windows Forms/Windows Presentation Foundation).

Porém as boas notícias são que as principais distribuições .NET estendem as bibliotecas CLI com equivalentes compatíveis com Microsoft de ASP.NET, ADO.NET e Windows Forms para fornecer plataformas de desenvolvimento totalmente equipadas e em nível de produção. Atualmente, existem duas grandes implementações da CLI (além da oferta específica para Windows da Microsoft). Embora este texto se concentre na criação de aplicações .NET utilizando a distribuição .NET da Microsoft, a Tabela 1-5 fornece informações com relação aos projetos .NET Mono e Portable.

Tabela 1-5. *TabelaDistribuições .NET Open Source*

Distribuição	Função
`http://www.mono-project.com`	O projeto Mono é uma distribuição open source da CLI voltada para diversas distribuições Linux (por exemplo: SuSE, Fedora e assim por diante), bem como Win32 e Mac OS X.
`http://www.dotgnu.org`	Portable .NET é outra distribuição open source da CLI que roda em diversos sistemas operacionais. Portable. NET é voltado para o máximo possível de sistemas operacionais (Win32, AIX, BeOS, Mac OS X, Solaris, todas as principais distribuições Linux e assim por diante).

Tanto Mono quanto Portable.NET fornecem um compilador C# que está de acordo com a ECMA, mecanismo de runtime .NET, amostras de código, documentação, bem como diversas ferramentas de desenvolvimento que são funcionalmente equivalentes às ferramentas que vêm com o .NET Framework 3.5 SDK da Microsoft. Além disso, Mono e Portable .NET são partes integrantes do VB .NET, Java, e compilador C.

■ **Nota** Falaremos sobre aplicações .NET entre plataformas utilizando Mono no Apêndice B.

Resumo

O objetivo deste capítulo era determinar o modelo conceitual necessário para este livro. Comecei analisando diversas limitações e complexidades encontradas nas tecnologias anteriores a .NET, e continuei com uma visão geral de como o .NET e o C# tentam simplificar o estado atual das coisas.

Basicamente, o .NET resume-se a um mecanismo de execução de runtime (`mscoree.dll`)