Raphael Simon, Emmanuel Stapf,
Christine Mingins, and Bertrand Meyer

Eiffel for E-Commerce under .NET

At its July 2000 Professional Developers Conference (PDC), Microsoft introduced a new development framework, .NET, providing a whole set of mechanisms to program Web and Web-aware applications. Presented by Microsoft officials as the company's most important innovation since the introduction of Windows at the first PDC in 1991, .NET makes an unprecedented effort at supporting many different languages through a Common Language Runtime. Heeding the lesson from Sun's failed attempt to impose a one-language-fits-all Java corset (as stated, e.g., by Scott McNealy: "*I don't understand why anybody would be programming in anything other than Java*")[1], Microsoft has recognized what every software manager and developer knows: Today's enterprise software world is multilanguage.

For more than a year prior to the official unveiling of .NET, Interactive Software Engineering (ISE) and Monash University collaborated with Microsoft to make a first version of ISE Eiffel available on .NET. During his keynote introducing the technology at the PDC, Chairman and Chief Architect of Microsoft, Bill Gates, invited one of the authors to present the new .NET Eiffel offering called Eiffel# (see Fig. 1). The other language featured in that session was COBOL, ported to .NET by Fujitsu Software. It is significant that these two languages represent, on one side, compatibility with the past's legacy, and on the other, readiness for the challenges of today's mission-critical enterprise applications.

We feel that Eiffel on .NET provides an ideal combination for companies wishing to take advantage of best-of-breed technologies in operating systems, Internet and Web infrastructure, software development methods, and development environments. In particular, the openness of Eiffel to other languages and environments combined with .NET's emphasis on language neutrality, makes the resulting product an ideal vehicle for building applications containing components in many different languages, with Eiffel serving as the "glue" between them. The first JOOP column in this series presented Eiffel as the "component combinator"[2]; Eiffel on .NET gives a new dimension to this role of Eiffel. We also see that the combination of Eiffel with .NET's Web mechanisms, especially Active Server Pages + (ASP+), provides remarkable tools for building Web and e-commerce applications.

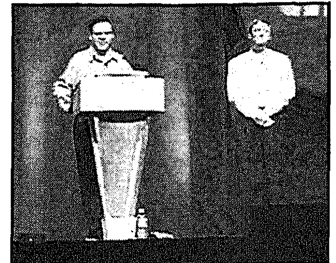

Figure 1. Bertrand Meyer and Bill Gates introducing Eiffel#.

This column describes the current state of the Eiffel# implementation, related Eiffel work on .NET, and future developments. A longer article written for MSDN before the PDC[3] provides additional background.

Raphael Simon is a senior engineer in charge of the Windows applications and tools division at ISE, Santa Barbara, CA. Emmanuel Stapf is a senior engineer at ISE and head of the compiler and environment division. Christine Mingins is Associate Professor and Associate Head of School at Monash University (Melbourne, Australia). Bertrand Meyer is CTO of ISE and a Professor at Monash University. They may be contacted at info@eiffel.com.

ABOUT THE .NET FRAMEWORK

The .NET framework is the next generation Web technology developed at Microsoft, which leverages many technical solutions for building Internet applications. A particularly important component for building Web-based applications in a considerably faster and easier way than more "traditional" approaches is ASP+.

The core of the technology consists of a runtime that interprets and/or compiles the .NET virtual machine's internal language, known as MicroSoft Intermediate Language (MSIL), with metadata. The metadata describes each component part of the system, including the prototype for all its methods, fields, or events. One of the most visible consequences of this strategy is the removal of any need for developers to write separate "glue" code to make components interact with each other. Unlike COM and CORBA, component-based development on .NET does not require any Interface Definition Language (IDL) modules. The components are just naturally able to interoperate thanks to the metadata. This is in line with Eiffel's philosophy as embodied in the Self-Documentation Principle[4]: Avoid multiple descriptions of the same software properties with all the resulting problems of tedious, error-prone development (Who really wants to write a module and then write its interface *again* in IDL?) and risks of divergence between these descriptions. Instead, include all the requisite information in the software, and rely on software tools to extract views at different levels of abstraction.

The unprecedented level of interoperability provided by the

.NET framework has some fascinating consequences for Eiffel users. At the PDC, and again at TOOLS USA 2000, we showed a debugging session within Visual Studio.NET that extended across languages to follow execution step-by-step from Eiffel classes, to classes written in C# (the native language introduced by Microsoft to program the framework directly) and back again to Eiffel. It is also possible to have Eiffel classes inherit from classes written in C++, C#, and other .NET languages. This means that Eiffel programmers can have immediate access to thousands of reusable library classes available in other .NET languages, not just as "consumers," but also as "extenders," taking advantage of the Eiffel mechanism to extend these components and integrate them into their own applications. The possibilities for reuse and interoperability are far ahead of any earlier technology.

PRODUCING .NET SYSTEMS FROM EIFFEL

Targeting .NET for a language compiler means being able to produce MSIL and the associated metadata. Generating MSIL would be enough if the aim was just to "compile Eiffel under .NET," but would fall short of our goal of providing a general-purpose framework for multilanguage interoperability because other languages would not be able to reuse Eiffel types without the metadata that describes them. One of the objectives set by ISE regarding the integration of Eiffel is the ability to reuse existing types written in any language, as well as the generation of types that can be understood by any other .NET development environment. Eiffel, as noted, is a .NET *extender*, meaning that you can write Eiffel classes that inherit from classes written in other languages, extend them, and then recompile them to IL, giving other environments the possibility of reusing the new type.

This generation of MSIL and metadata is done through a special switch in the Eiffel Ace file (the compilation control file) so that the new Eiffel compiler is fully integrated into ISE's IDE, EiffelBench. Existing Eiffel programmers will be able to work exactly as they did before the integration. As noted, ISE Eiffel will also be available under Microsoft's Visual Studio.NET. This integration will help new Eiffel developers that do not want or do not have time for learning a new environment, but already make use of Visual Studio to shorten their learning curve. The integration will support all the specific functionalities of Visual Studio such as syntax highlighting, the debugger, wizards, etc.

Another key goal is the ability to write ASP+ pages in Eiffel. This is available today and is described in a later section.

Strategy

With these goals in mind, ISE organized the integration around two major milestones. The first step of the integration has led to the implementation of a new language called Eiffel# (pronounced Eiffel sharp). This is an almost complete subset of Eiffel specifically designed to target .NET that embodies the full extent of Design by Contract and genericity. The only major construct not supported in Eiffel# is multiple inheritance from effective (non-purely-abstract) classes. Eiffel#'s object model already goes beyond the model supported by most OO languages (Java, for example, has neither genericity nor Design by Contract) and is more than sufficient to build real applications while keeping the native object model of .NET.

This step has been completed and Eiffel# is available today from ISE. The second step, currently in progress, will extend the results to encompass the full object model of Eiffel.

Both the integration into Visual Studio and ASP+ started with Eiffel#. The support for ASP+ started with the support for Web services.

The release schedule, including support for full Eiffel, will be synchronized with the schedule for successive beta and commercial releases of .NET.

Web services in Eiffel: ASP+ and .NET

One of the most exciting aspects of the technology is the ability to develop Web services in Eiffel. We encourage readers to see the interactive demo online at ISE's Web site.[5] It will enable you to register (fictitiously) to the TOOLS conference, then see the list of registrants as if you were one of the conference organizers. A particularly interesting feature is the use of Eiffel contracts for the testing and debugging of Web and debugging applications. If you enter wrong information (see Fig. 2), you will get a contract violation.

The Web user did not fill in the "last name" entry, which a precondition requires to be nonempty. Clicking on *Register* will yield the following page (see Fig. 3):

You wouldn't, of course, expect to show this page to a normal user of the site. But for an e-commerce site such as this one, developers' contracts provide a remarkable tool for specification, documentation (as in the rest of Eiffel development), and, as shown here, for testing, and debugging.

Another attractive aspect of this demo is that it can instantaneously be turned from a Web application running in a browser, into a client-server application running on a Windows client out-

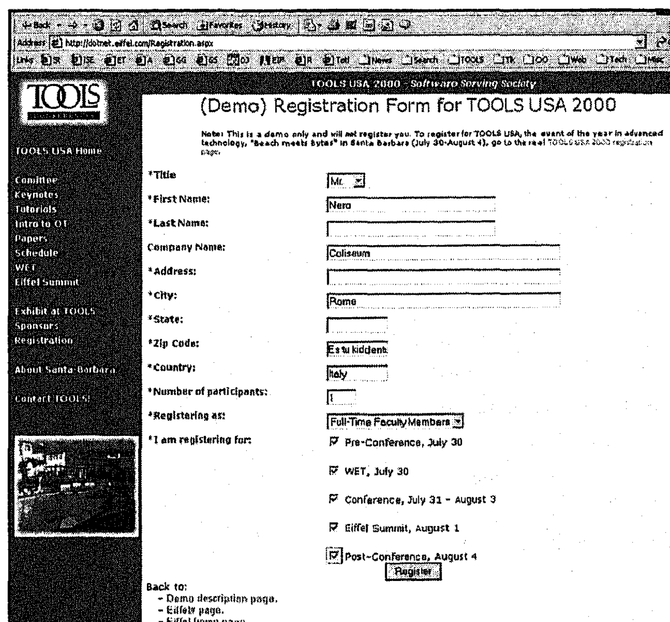

Figure 2. Filling in an Eiffel#-ASP+ form (erroneously).

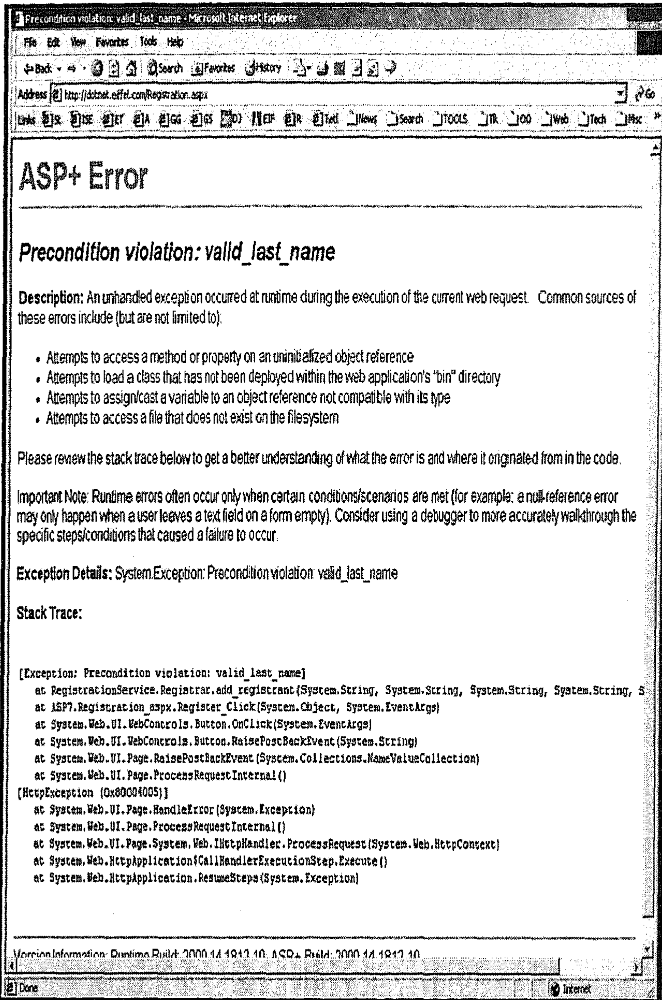

Figure 3. A contract violation for distributed objects.

side of the Web. This is one of the common challenges for enterprise applications: how to provide both modes of operation, Web and non-Web, without extensive reprogramming.

The full source text of the demo (Eiffel and ASP+) is available from the demo page. As you will see, it includes, in its current form, a small amount of C# "glue" code in the ASP+ page. This need will soon go away as we implement Eiffel as an ASP+ language by providing a `'@language="Eiffel"'` directive.

Eiffel# vs. Eiffel

A key requirement for Eiffel# is the exclusive use of the .NET runtime without any other Eiffel-specific runtime. The most challenging issue is support for multiple inheritance since the .NET runtime was designed to support single inheritance only. Because Eiffel# must only use the .NET runtime, it has to follow the .NET object model and thus disallows multiple inheritance of effective or partially deferred classes. You may, however, multiply inherit, pure-deferred classes.

Eiffel# does not support several of the new Eiffel constructs that were added after the publication of the current edition of *Eiffel: The Language*.[6] These constructs include agents and related classes, generic conformance, and generic argument creation.

The last difference between the two languages lies in the semantics of expanded types. Expanded types in the .NET environment are directly mapped to the so-called *value types*. Although fundamentally the same, Eiffel expanded types and .NET value types do not behave in the exact same way. In particular, value types are *sealed*, meaning that one cannot inherit from them. As a result, you cannot inherit from expanded types in Eiffel#.

These are the only differences between Eiffel# and Eiffel. In an earlier presentation,[3] we mentioned that Eiffel# did not support Eiffel's covariance for argument types. Since then, however, we have implemented covariance in the normal Eiffel style.

Other than multiple inheritance, Eiffel# supports all the hallmarks of Eiffel programming: Design by Contract, exception handling, and genericity. Genericity came as a pleasant surprise—we had initially announced it would not be in Eiffel# but found a way to implement it in time for the first release.

.NET Specifics

While in a standard environment, providing that the choice between building a Dynamic Link Library (DLL) or a Windows Executable (EXE) is enough, .NET defines new concepts such as assemblies and modules that any compiler targeting the environment should support. An assembly is made of a group of modules, either a DLL or an EXE, and corresponds to an application. For that reason, the Ace file for Eiffel# introduces new options to describe the different modules that will be part of the assembly. The Eiffel# compiler generates one assembly whose name is the name of the system as given in the ACE file. You may specify whether the assembly should be an EXE or a DLL in the `il_generation` default option as follows:

```
system
        sample
root
        ROOT_CLASS: "make"
default
        il_generation ("exe") -- "dll" to generate a DLL
...
```

In this example, the compiler generates a single file `"sample.exe"` containing both the assembly and the module. In case you would like to specify different files for multiple modules, you can use the `module` option for each cluster and override the option for any class in the cluster:

```
system
        sample
root
        ROOT_CLASS: "make"
default
  il_generation ("exe") -- "dll" to generate a DLL
cluster
        root_cluster: "c:\my_app"
           default
              module ("my_app")
```

```
                    option
                       module ("root"): ROOT_CLASS
                    end
    ...
```

This Ace file defines three modules:

- The first module, which includes the assembly manifest, is "sample.exe".
- The second module, "my_app.dll", includes all classes in cluster *root_cluster* except the class *ROOT_CLASS*.
- The last module, "root.dll", includes the class *ROOT_CLASS*. This mechanism allows you to define as many modules as you need and group the classes of the Eiffel system the way you want to.

Another feature specific to .NET is the notion of namespace. Any .NET type is associated with a namespace that ensures the uniqueness of the type name in the system. You can define a default namespace for all the classes of the Eiffel system by using the following default Ace option:

```
system
            sample
root
            ROOT_CLASS: "make"
default
  il_generation ("exe") -- "dll" to generate a DLL
  namespace ("MyApp")
...
```

In this example, all the classes of the Eiffel system will be generated in the namespace "MyApp.<cluster_name>" where <cluster_name> is the name of the cluster that contains the class. You may override the default namespace for each cluster as follows:

```
system
            sample
root
            ROOT_CLASS: "make"
default
  il_generation ("exe") -- "dll" to generate a DLL
  namespace ("MyApp")
cluster
            root_cluster: "c:\my_app"
               default
                  module ("my_app")
                  namespace ("Root")
               option
                  module ("root"): ROOT_CLASS
               end
...
```

With this ACE file, all the classes that are part of the cluster *root_cluster* will be generated in the namespace "Root". Note that the name specified in the cluster clause is not appended to the namespace defined in the default clause. Finally, the Eiffel# class might include an **alias** clause (see the External Classes section for a description of the **alias** keyword) in which case the name specified in the clause overrides any namespace specified in the ACE file.

Another major difference coming from the dynamic nature of the .NET environment is how contracts behave at runtime. In a "classic" environment, a contract violation results in a raised exception and the level of assertion checking is decided at compile time. This approach is no longer satisfactory in .NET where the caller of a contracted routine might be in a different module. The client of a contracted component should be able to decide which level of contract checking should be set, if any. This is the reason for having a standard interface implemented by contracted components that defines the possible contract interpretations. The interface called *IContract* defines routines to set the level of contract checking, similar to those of the Eiffel compiler: preconditions only; pre- and postconditions; preconditions, postconditions, and invariants.

IContract also allows specifying whether to raise an exception in case of a contract violation.

```
interface IContract
{
     // subject to modification
     public bool precondition_activated;
     public bool postcondition_activated;
     public bool invariant_activated;

     public void enable_precondition();
     public void enable_postcondition();
     public void enable_invariant();

     public void disable_precondition();
     public void disable_postcondition();
     public void disable_invariant();

     public bool exception_on_violation();
     public void enable_exception_on_violation();
     public void disable_exception_on_violation();

     public ContractException last_contract_exception();
}
```

The Eiffel# compiler will automatically generate an implementation of the *IContract* interface with the default contract-checking level specified in the ACE file:

```
system
            sample
root
            ROOT_CLASS: "make"
default
  il_generation ("exe")
  assertion (require) -- can be "no", "require",
                      -- "ensure" or "invariant"
...
```

If you omit the `assertion` option then *IContract* is not generated. If you choose the option `"no"` then *IContract* is generated but no contracts will be checked until a client activates contract checking.

LEVERAGING THE .NET FRAMEWORK

We have seen how you can use Eiffel# to build .NET components. Since the compiler generates all the necessary metadata, other languages can reuse the Eiffel# components in any way they like (heritance or client relationship). The next question is "how do I reuse existing components in Eiffel#?" Existing components cover the Microsoft libraries as well as components written by other parties.

Strategy

ISE will provide Eiffel# libraries that wrap the Microsoft framework. These libraries include wrappers for the *Base Class Library, WinForms,* and *WebForms.* The Base Class Libraries include the definition of the basic types such as collections, remoting services, threading services, security, IO access, etc. needed for any system.

The WinForms classes are a wrapper around the Win32 APIs that provides a clean object model to build a GUI in .NET. WebForms also provide a way to build GUIs, but on the Web. They include types like *DataGrid* or *HTMLImage.*

These three libraries are distributed with Eiffel# so you can reuse their classes directly in your system.

The Emitter

Obviously, the Microsoft libraries are not the only .NET components you might want to access. Your system may require the integration of hundreds of components written in various languages. For this reason, ISE provides a tool called *Emitter* that can analyze any .NET assembly and produce an Eiffel wrapper for every type it defines. The Emitter accesses the metadata bound into each type defined in the assembly, maps them into an Eiffel equivalent, and generates the corresponding classes.

Although an assembly may include references to other assemblies (called *external* assemblies), the emitter will only generate classes for the types defined in the given assembly. This avoids, for example, generating the Base Class Libraries for all the assemblies you need to wrap (since almost any assembly has a reference to the Base Class Libraries).

Because any public .NET type must comply with the *Common Language Specification* (CLS), and because the CLS differs in certain aspects from the Eiffel model, the Emitter has to perform a few nontrivial transformations to map the .NET types into Eiffel classes. Perhaps the most important mismatch is the inclusion of overloading into the CLS. The Eiffel model prohibits overloading and requires disambiguating any overloaded function. The Emitter uses a clearly understandable algorithm for this purpose. Details of the algorithm may be found in the MSDN article.[3] As an example, for the following C# functions

```
public static void WriteLine (String format,
        Object arg0);
    public static void WriteLine (int value);
    public static void WriteLine (String value);
```

the Emitter will generate the Eiffel functions:

```
WriteLine_System_String_System_Object

        (format:STRING; arg0: ANY)
    WriteLine_System_Int32 (value: INTEGER)
    WriteLine_System_String (value: STRING)
```

Table 1 lists all the primitive types as defined in the CLS and their Eiffel equivalent:

Note that some of the Eiffel types (sized integers and reals) were recently added to the Eiffel Kernel Library.

External Classes

The Eiffel# classes that the Emitter generates do not include any logic; they are just needed for the Eiffel-type system. This means that the Eiffel# compiler does not generate any IL code for these classes. Any calls to functions on classes generated by the Emitter are direct calls to the .NET type; there are no indirections and thus no performance penalty. For the compiler to recognize such classes, ISE introduced a new mechanism in Eiffel called *external* classes. Such classes can only include external features, i.e., features that are not written in Eiffel, but rather methods or functions on an already existing .NET type. All the features of an external class should be features belonging to the same .NET type. You can declare such a type with the following syntax:

```
frozen external class
        SYSTEM_CONSOLE
alias
        "System.Console"
...
```

The string that follows `alias` contains the name of the .NET type that the Eiffel class wraps. Since .NET types might be *sealed,* i.e., they might forbid other types to inherit from them, and since such a concept does not exist in Eiffel, Eiffel# introduces a new use for the Eiffel keyword `frozen`. You may use `frozen` in front of `external` to tell the Eiffel# compiler that no Eiffel# class should inherit from the .NET type.

The external class should then list all the features you need to access. All of these features are external features. The syntax for an external .NET feature is the following:

```
frozen ReadLine: STRING is
      external
         "IL static signature :System.String use
System.Console"
      alias
         "ReadLine"
      end
```

where `frozen` indicates that the feature may not be redefined in a descendant (You may redefine external features if they are *virtual,* in which case `frozen` should not be used), `ReadLine` is the Eiffel# feature name, and `STRING` is the return type of the feature. The string

Table 1. Primitive types as defined in the CLS, and their Eiffel equivalents.

CLS Primitive Type (Description)	Eiffel Type
System.Char (2-byte unsigned integer)	CHARACTER
System.Byte (1-byte unsigned integer)	INTEGER_8
System.Int16 (2-byte signed integer)	INTEGER_16
System.Int32 (4-byte signed integer)	INTEGER
System.Int64 (8-byte signed integer)	INTEGER_64
System.Single (4-byte floating point number)	REAL
System.Double (8-byte floating point number)	DOUBLE
System.String (a string of zero or more characters; null is allowed)	STRING
System.Object (the root of all class-inheritance hierarchies)	ANY
System.Boolean (True or False)	BOOLEAN

that follows the **external** keyword specifies the kind of external, the .NET function signature, and the .NET type on which the function is defined. The string following the **alias** keyword contains the .NET name of the function. There are different kinds of external features depending on the type of method they provide access to. Eiffel# defines seven new kinds of externals listed in Table 2.

You can define such external features in nonexternal classes should you need to. In the special case of external classes, the .NET-type name appearing at the end of the string following the **external** keyword should be the same as the one that appears after the **alias** keyword following the declaration of the class (see previous code section).

The external features can be called from clients or descendants of the class the same way you would call any other Eiffel feature. So, if your system includes a feature that needs user input, it can include the following code (which does not apply the usual Eiffel command-query separation style):

```
need_user_input is
    -- Take user input and do stuff.
  local
    io: SYSTEM_CONSOLE
    input: STRING
  do
    create io.make
    input := io.ReadLine -- calls System.Console.ReadLine()
    -- do stuff
  end
```

However, because *ReadLine* is a static external, you do not need to instantiate the wrapper to call it, so the following code

```
need_user_input is
    -- Take user input and do stuff.
  local
    io: SYSTEM_CONSOLE
    input: STRING
  do
    -- removed creation of io
    input := io.ReadLine -- calls System.Console.ReadLine()
    -- do stuff
  end
```

is sufficient to make the external call. This is also valid for static field access and static field setting. Other kinds of externals do require the wrapper to be instantiated.

Eiffel# Libraries

The Eiffel# compiler incorporates an Eiffel# Base Class Library, which follows the design principles found in the standard EiffelBase Library. This library makes extensive use of genericity and contracts to provide a clean and powerful set of data structures you can reuse in your system. It also uses the external classes provided with Eiffel# that wrap the .NET Base Class Library.

The two other sets of classes provided with the Eiffel# compiler wrap the WinForms and WebForms .NET libraries so that you can easily build GUI and WEB applications.

The .NET Contract Wizard

As part of this development we produced a new tool, the .NET Contract Wizard, that enables users through the metadata mechanism to interactively add Eiffel-like contracts to .NET components coming from arbitrary languages. This tool will be described in detail in a separate column but is already available for developers who want to apply the benefits of Design by Contract in languages other than Eiffel. This important extension was made possible by the metadata facilities of .NET.

CONCLUSION

The aim of this project and the resulting products is to provide a full integration between ISE Eiffel and the .NET environment. The combined power of the platform and the development environment should yield the dream environment for building the powerful Internet applications that society expects from us today. Eiffel on .NET provides flexibility, productivity, and high reliability. It is impossible to overestimate the benefits of Design by

Table 2. Seven new kinds of externals defined by Eiffel#.

.NET Function Kind	Eiffel External
Method	"IL signature … use class_name"
Static Method	"IL signature … static use class_name"
Field Getter	"IL signature … field use class_name"
Static Field Getter	"IL signature … static_field use class_name"
Field Setter	"IL signature … set_field use class_name"
Static Field Setter	"IL signature … set_static_field use class_name"
Constructor	"IL signature … creator use class_name"

continued on page 58